Broadview
www.broadview.com.cn

两打程序员，3年时间，4732个bug，只为打造超卓软件



DREAMING IN CODE

梦断代码

【美】Scott Rosenberg 著　韩磊 译

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

DREAMING IN CODE

梦断代码

【美】Scott Rosenberg 著　韩磊 译

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

软件乃是人类自以为最有把握，实则最难掌控的技术。本书作者罗森伯格对 OSAF 主持的 Chandler 项目进行田野调查，跟踪经年，试图借由 Chandler 的开发过程揭示软件开发中的一些根本性大问题。

本书是讲一事，也是讲百千事；是写一软件，也是写百千软件；是写一群人，也是写百千万人。任何一个在软件领域稍有经验的技术人员看完本书，必掩卷长叹：做软件难。

Copyright©2007 by Scott Rosenberg
This edition arranged with Stuart Krichevsky Literary Agency
All rights reserved.
Authorized translation from English language edition published by Crown Publishers, an imprint of the Crown Publishing Group, a division of Random House, Inc., New York.

本书中文简体版专有出版权由 Stuart Krichevsky Literary Agency 授予电子工业出版社，未经许可，不得以任何方式复制或抄袭本书的任何部分。

版权贸易合同登记号 图字：01-2007-2963

图书在版编目（CIP）数据

梦断代码 /（美）罗森伯格（Rosenberg, S.）著；韩磊译. —北京：电子工业出版社，2008.6
书名原文：Dreaming in Code：Two Dozen Programmers，Three Years，4,732 Bugs，and One Quest for Transcendent Software
ISBN 978-7-121-06679-5

I. 梦… II. ①罗… ②韩… III. 软件开发－通俗读物 IV. TP311.52-49

中国版本图书馆 CIP 数据核字（2008）第 068390 号

责任编辑：江 立
印 刷：北京智力达印刷有限公司
装 订：北京中新伟业印刷有限公司
出版发行：电子工业出版社
北京市海淀区万寿路 173 信箱 邮编 100036
开 本：787×1092 1/16 印张：21.5 字数：274 千字
印 次：2008 年 6 月第 1 次印刷
印 数：5000 册 定价：49.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

推 荐 序 一

大多数技术书籍只是讲技术和理论，但我们不知道在真实的软件开发过程中，这些技术和理论究竟是被什么样的人如何去使用?

《梦断代码》向我们展示了硅谷一流软件开发者是如何进行产品开发的，把真实的人、事、技术以及产品的发展过程结合在一起，每个有志于开发畅销产品的程序员都值得耐心去品味这个故事。

CSDN 总裁　蒋涛

2008 年 6 月

认证新题库
XINTIKU.COM

推 荐 序 二

软件技术日新月异，软件项目领导的艺术也是日新月异。如同开发者在不断把自己的代码当作艺术品来雕琢一样，组织好一群天才开发者，则更是艺术中的艺术。

软件开发者不是堆砌代码的工人，也无法安于命令式的任务布置。这些天才开发者们，低调、寡言、有独立的自我意识，他们并不迷恋于成为焦点的感觉，但却十分沉迷于自己认为伟大的创造。天才们在一起的合作，貌似创意无穷，实则合力有限；貌似独当一面，实则整合艰难。等等诸多的问题，似乎成了很多国内外软件企业的通病。

真正的领导者，实际上早已被要求有着化不可控为可控、化不同愿景为共同愿景、激发天才们最大潜能的能力。这对于整个项目而言，这样的图景绝不应该止于一个美好愿望，尽管难上加难，但在所有开发者和经理们的努力下，我坚信一定可以实现。

认证新题库
XINTIKU.COM

Discuz!创始人 康盛创想（北京）科技有限公司总裁 戴志康

2008 年 6 月

作 者 的 话

这世上有太多给软件开发者看的指导书。这本书并不是其中之一。我自己只是个入门级程序员，怎能在专家们的面前班门弄斧呢。而且，如果我的研究在构造更好软件方面发掘出了前所未知的创新或洞见，何不径自去找投资商，何苦还要呈示给读者。

我期望程序员喜欢这本书，但本书对其他人也不无益处。它提出问题，讲述故事。为什么好软件如此难做？因为看来无人能有确定答案，即便是现在，21 世纪伊始之际，计算机时代开始 50 年之后，我以亲涉踏勘的方式写出这个做软件的故事——关于一队人马并肩托起代码大石、欲将其推上山顶，虽历经磨难，但仍奋力创造某种有用、丰富且持久之物的故事。

司各特·罗森伯格（Scott Rosenberg）

认证新题库
XINTIKU.COM

目　录

第 0 章　软件时间 …… 1
第 1 章　死定了　[2003 年 7 月] …… 11
第 2 章　Agenda 之魂　[1968 年～2001 年] …… 31
第 3 章　原型与 Python　[2001 年～2002 年 11 月] …… 53
第 4 章　乐高王国　[2002 年 11 月～2003 年 8 月] …… 77
第 5 章　管束奇客和狗　[2003 年 4 月～8 月] …… 107
第 6 章　完成设计方案　[2003 年 7 月～11 月] …… 133
第 7 章　细节视图　[2004 年 1 月～5 月] …… 161
第 8 章　白板上的即时贴　[2004 年 6 月～10 月] …… 191
第 9 章　方法 …… 217
第 10 章　工程师和艺术家 …… 249
第 11 章　通往狗食版之路　[2004 年 11 月～2005 年 11 月] …… 285
尾声　长赌　[2005 年～2029 年及以后] …… 317
译后记 …… 327
附录 A　专有名词对译表 …… 330

认证新题库
XINTIKU.COM

第 0 章　软件时间

那是 1975 年的冬天。我在终端机房中俯身敲击一台电传打字机，每打完一行，那笨重的机头就会摇头晃脑猛然撞回最左边，开始新的一行。我从几个小时前开始输入一行行黑代码[①]，忘记了时间流逝，全然不知已是午夜时分。看门人已经关闭廊灯。我并没有得到许可在纽约大学物理系大楼中流连忘返、使用向高中学生免费发放的计算机账号。不过，倒也无人责难。

那时我年方十五，正迷恋于一个叫做 **Sumer** 的游戏，在游戏中，我管理着新月沃土[②]上一座古代城邦。今天的电脑游戏玩家也许会嘲笑其稚嫩：它在一卷纸上逐行打印出大写字母，报告游戏进程。玩家运筹帷幄，分配食用和留种的谷物，然后程序就会告知城邦每年的发展情况。“汉谟拉比陛下，”程序像一个诚惶诚恐的宰相般报告说，“微臣伏启圣鉴……”

没过几天，我就已经把游戏玩了个遍。但是，和现在令青少年着迷的大多数游戏不同，**Sumer** 可以让玩家打补丁。谁都能够窥探其内部运行机制：该游戏只是向计算机发出的一系列简单指令，这些指令存储于一卷多行八孔纸带上。（电传打字机旁的塑料盘中堆积的纸带，几乎带来和游戏一样多的乐趣。）纸带像地下出版物一般在朋友间流传共享。只要

[①] Black code，指原来程序逻辑中没有的部分，采用类似侵入的手段加进去的代码。

[②] 新月沃土系自中东两河流域及附近一连串肥沃的土地。包括今日的以色列、西岸、黎巴嫩、约旦部分地区、叙利亚，以及伊拉克和土耳其的东南部。由于在地图上形似一弯新月，所以美国芝加哥大学的考古学家詹姆士·布雷斯特德（James Henry Breasted）把这一大片肥美的土地称为“新月沃土”。

认证新题库 XINTIKU.COM

花几个钟头学点简单的 Basic 语言，改游戏就会和玩游戏一样容易：将纸带上的指令装载到计算机，然后开始往程序里加代码。

Sumer 是个空白画板——历史只是个轮廓，随时准备着让少年的梦想来浇筑。我和朋友们掌握了它简单的构造，开始往里加东西。让玩家可以选择不同的宗教信仰吧！偶尔来一次腺鼠疫，会发生什么事？蛮族入侵者应该很酷。嘿，搞几具弹石机如何？

那天晚上，我倾力于改造游戏中民众造反模式的设计。Sumer 只提供粗糙的起义模式；如果你干得太差，人民就会起来推翻你。（Sumer 的原作者是个乐天派。）我认为，游戏中的起义模式应该多种多样，所以就创建了一些子程序补丁——让 Sumer 陷入内战，或是引入一个想要争取合法地位的反对派政府。

我不在乎干到多晚。F 线地铁整夜运行，可以把我带回位于皇后区的家。一定得重新设计游戏中的革命模式！

1010101010101010110101101110

四分之一世纪后，2000 年 5 月，我坐在旧金山的一间办公室里，眼睁睁紧盯着一台现代计算机屏幕（高解析度、数百万种颜色）。喝剩的咖啡纸杯胡乱摆在键盘边上。正是凌晨 5 点。

认证新题库 XINTIKU.COM

那时我四十岁，是在线杂志 Salon 的创始人和总编，此外还负责一个软件开发项目。我们花了几个月精心规划，希望给网站增加动态特性，使之彻底改观。然而，现在我却眼看着项目濒临绝境。

在没日没夜苦干了几个星期之后，主力程序员终于宣告工作完成，自己要飞往夏威夷，度一个全家盼望已久的假期。剩下他的老板，技术副总裁查德·迪克森（Chad Dickerson），独自琢磨为什么存储网站文章的数据库就是不肯与负责显示页面的程序对接。查德两个通宵没合眼，努力修复问题。若是不然，到周一早上，我们的两百万读者就只能看到

网站上没更新过的旧闻了。

难道我们以前没做过软件吗?

做过。

是没有全面测试吗?

显然不太充分。

怎么会搞得一塌糊涂?

鬼晓得。

我吃完了自动售货机里最后一袋饼干，徘徊又等待，却仍是毫无指望。时间还多。还有时间去读那位以新项目的名义准备香槟加蛋糕聚会的倒霉同事的邮件，回复他说："或者咱们再等等吧。"还有时间去体会身陷困境孤立无援的感受，然后琢磨将系统的中心服务器命名为"卡夫卡"是不是个好主意。

大约早晨 9 点，我们终于发布了站点"改进版"的第一个版本。又是周一清晨，其他同事相继出现在办公室，他们过了好一会儿才知道，原来我们六个人昨夜压根儿就没回家睡觉。

又过了几星期，程序员们修复了最严重的问题，软件运行趋于平稳。但后来每每听说某公司打算"升级其软件平台"、重新搭建一套大型系统时，我总不免暗自担心。

认证新题库
XINTIKU.COM



20 世纪 90 年代科技行业的兴盛，给我们带来了"互联网时间"的概念。对该短语含义的理解见仁见智，但多指"快速"之意。数字时代的新时间机制下，一切皆有可能发生——技术产生、公司创立、创造财富——而且速度惊人。这意味着你没时间做到尽善尽美——无须担心，因为别人也一样。

随着投资潮退，“互联网时间”这个短语也风头尽失，很快被其他时髦词所代替。但新词掮客们的确一语中的。在做软件的过程当中，时间似乎确实时快时慢。如果一切顺利，你会沉浸在心理学家称之为“流逝”的状态中，全然忘记了时间。如果事有不谐，你又会陷入困境，四顾茫然、举步维艰。无论是哪种情况，时钟都被抛诸脑后。你用的是软件时间。



在使用一门新编程语言时，程序员的第一个程序通常是“Hello World”——输入一系列代码，召唤计算机，命令它打印出这两个词，向主人致敬。在 Sumer 游戏所用的 Basic 语言中，这个程序像这样：

```
10 PRINT "HELLO WORLD!"
20 STOP
```

“Hello World”程序一无所用，但足可蛊惑人心；它鼓励新手，唤起每个程序员心中乐观的一面。**既然能叫它说话，就能让它做任何事**！计算机器协会（The Association for Computing Machinery），计算机领域中的 ABA[①]或 AMA[②]，维护了一张网页，上面列出将近两百种编程语言版本的“Hello World”程序。简直就是程序代码的罗塞塔石碑[③]。

在 Java 这种商业世界中流行的重量级编程语言里面，“Hello World”看起来高不可攀：

```
class HelloWorld {
    public static void main (String args[]){
        System.out.println("Hello World!");
        }
    }
```

① American Bankers Association，美国银行家协会

② American Medical Association，美国医药协会

③Rosetta Stone，1799年法国士兵在埃及发掘出来的象形字黑石碑。也是美国著名的外语教程品牌。

Public static void：无数个 Java 程序代码块中，都有这串密语存在。这串词有特别的技术含义。不过我常常把它看作一阙机器诗篇，在它召唤出的冷宫里面，多少软件项目一开始雄心勃勃，最终却未结善果。



如果你和计算机编程打过交道，就很难不对它又爱又恨。作为少年游戏玩家，我品味过编写代码的巨大快乐。作为媒体工作者，我见证了软件世界中无数个悲惨故事——无论是跨国公司、政府机构，还是军工大鳄，都曾一头撞上过代码的冰山。而作为一个经理人，我也得对付自己桌面上的泰坦尼克号。

这 25 年令人气馁的软件历史，也许不具代表性，但却是我的个人经验。依照硅谷的数字乌托邦理想，事情理应朝好的方向发展。在 Salon 网站发布失败后的几个月里，理想与现实之间的差异开始对我露出了利齿獠牙。



图 0-1　Salon 网站

编程已不再处于萌芽期。我们的世界依赖于无穷复杂的软件。在长达半个世纪的研究和实践之后，为什么还是很难做到按时限、按预算做出计算机软件？为什么还是很难让软件可靠而安全？为什么还是很难把软件做得易于学习使用，且具备按需修改的灵活性？这只跟时间和经验有关吗？是否有出现某种根本性突破的可能？在软件的本质特性（抽象性、复杂性及延展性）上，是否存在某种总能击倒我们的无常之物，将开发者咒入充满不可挽回的延误和根深蒂固的缺陷的世界？

“软件难做”，编程界经典教科书的作者高德纳（**Donald Knuth**）这样写道。但原因何在？



你可能已经注意到，我把本章标为“第 0 章”。我无意搞笑，只是想指出计算机程序员和其他人的一处小小不同：程序员从 0 开始计数，而不是从 1 开始。要解释这种习惯的来源，得从计算机中央处理单元里的寄存器，以及数据队列的结构等等奥义秘辛说起。不过，我发现最直截了当的解释来自于一个网页，该网页试图向大众解释黑客的行为——“黑客”一词的本义是“痴迷的编程匠人”，而非后来衍化出的贬义“数码入侵艺术家”。

为什么程序员要从 0 开始计数？**因为计算机从 0 开始计数**！所以，程序员也训练自己这样计数，以免让他们要指示操作的计算机产生误解。这本也无伤大雅，只是使用计算机的大多数人是从 1 开始计数，未免令人烦恼。往下到系统层面，在这个层面上，数据被存储和操作——意味着我们的金钱、工作和设想被转换为机器可读的符号——计算机程序及编程语言经常会做小小的偏移操作，即“+1”或“-1”，使得计算机从 0 开始计数的列表与人类从 1 开始计数的列表保持同步。

在计算机的二进制数字世界里，所有的信息都被简化为 0 和 1 的序列。但是，在 0 和 1 之间有空间存在，在机器计数和思考的方式与人类计数和思考的方式之间也有空间存在。当你寻找软件缺陷、延误和不按设计思路运行的原因时，那原因就藏身于这空间之中。



在构思本书的那段时间里，我每天要驾车从旧金山海湾大桥（Bay Bridge）上通过。一天早晨，当我的车努力爬上连接奥克兰（Oakland）岸边和桥东段中心较高地带的长长引桥时，我发现，右边有个新物体挡住了海湾碧水和远山绿树：那是一台高耸的红色起重机的顶端，正好超出桥面。它在那儿日复一日地矗立着，突然有一天，又多了 12 台起重机，在桥北一线齐齐排列，如同挤在食槽旁的机械怪兽，等着倒霉的上班人士送进嘴来。

图 0-2　建设中的海湾大桥

这工程是要替换双层大桥的北半部分。在 1989 年的 Loma Prieta 大地震发生时，该部分上层一段五十英尺长的桥面坍塌到下层的车行道上。现在，将在旧桥旁边搭建一座更安全、更现代的新桥。

随后几个月，这些 240 英尺高的起重机，开始将一根根直径达 8 英尺、长达 300 英尺的锈钢管打进海湾水底。在清晨时分，从我远在伯克利（Berkeley）山的家中都可以听到敲击声。总共将会有 160 根这种大管子被打入海底，填上混凝土，支撑新桥的水上部分。整个过程设计精密、执行无误；它分毫不差，完全满足了我们对**工程**一词的信心。

关于软件缺陷的话题，只要谈上几分钟，必会有人拍案叹道，“为什么就是不能像造桥那样造软件？”



和摩天大楼、水坝等永久性建筑一样，桥梁体现了人类对物理世界的技术把握。在过去半个世纪里，软件成为构建这个世界的虽不可见但却深入渗透的人造物。“人类文明运行于软件之上”，广为应用的计算机语言 C++发明人比昂纳·斯卓思柯普（Bjarne Stroustrup）这样说道。

初听起来，这像是奇谈怪论或是自卖自夸。即便没有 Microsoft Windows，人类文明也会同样延续，对吧？然而，软件并不只是用来发电子邮件或写报告的程序那么简单；它已经不声不响地渗透到生活的每个角落。它存在于厨具里、汽车中、玩具里、建筑中。商业和银行、选举和新闻媒体、电影和交通网、医疗和国防、科研和基础公共服务——人类生存之所需都系于计算机代码这根易断的细线上。

而且我们要为其脆弱埋单。根据国家标准和技术学会（National Institute of Standards and Technology）2002 年的研究，软件错误每年造成美国 595 亿美元的经济损失，三分之二的项目明显延误或超出预算，甚

至干脆无疾而终。

人类文明运行于软件之上。但是，软件创建艺术却隐于暗处，即便对于专家们也是如此。在历史上，我们从未如此地完全依赖于这样一种人类自己不知道怎么才做得好的产品。在对软件系统的加速依赖和踱着方步学习怎么做好软件之间，有一条巨大且有时叫人恐惧的壕沟。对软件的依赖以指数级增长，而做软件的技能——和应用技能的愿望——却进展缓慢。

你要和程序员说这些，就等着挨批吧。这边厢，有人也许会说，世界从未如此光明：我们拥有了比以往更好的工具、更好的测试、更好的语言，还有更好的方法！那边厢，你又听人家说，自计算机时代的黎明以来，其实并未取得多少进展。计算机先驱莫瑞斯·威尔克斯（Maurice Wilkes）[①]回忆起 1949 年他在英国剑桥工作的情形，在拖着打孔纸带上楼给雏形计算机 EDASC 装载程序（如图 0-3 所示）时，他看到了未来："我强烈地意识到，生命中剩下的好日子，都将耗费在给自己的程序找错误上头。"从威尔克斯的时代直到现在，尽管有那许多创新，程序员却一直陷于调试除错之苦境。工作中只有百分之一的灵感迸发，剩下的是艰难寻找、汗湿重衣；他们的作品永远尚未完成或未臻至善，区别仅仅是"问题更少"的程度不同罢了。

[①]1967 年图灵奖获得者，他领导建造的拥有内存储程序的计算机 EDASC 于 1949 开始运行。

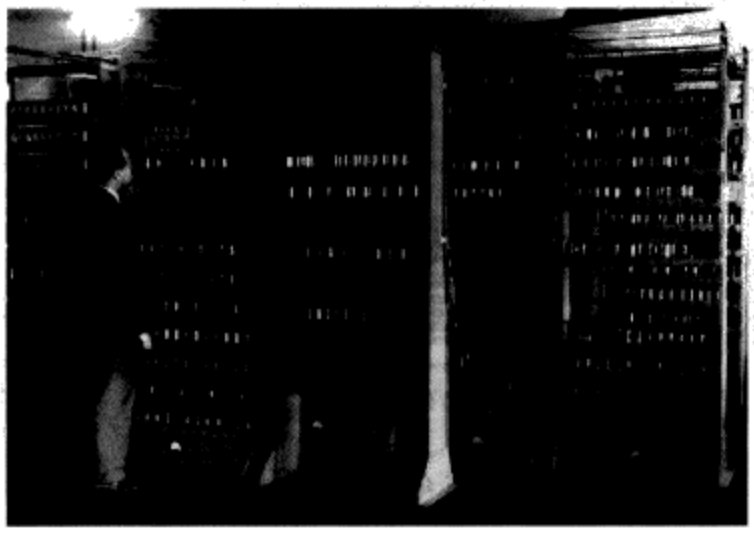


图 0-3　莫瑞斯·威尔克斯领导制造的 EDASC 计算机



弗里德里克·布鲁克斯

1010101010101010110101101110

软件就是麻烦一堆。而且我们不能够也不愿意把电脑一关走为上计。给我们带来挫败和束缚的软件，也用更多功能、更快更好的工作与生活方式来引诱我们。无路可回。我们对软件的需要，远甚于对它的仇恨。

所以我们梦想着得到更新更好的东西。在现代软件研究领域多有建树的专家弗里德里克·布鲁克斯（Frederick Brooks）在 1987 年写了一篇题为《没有银弹（No Silver Bullet）》的著名论文。布鲁克斯在论文中称，无论编写计算机程序是如何地令我们倍感挫败，也永远无法找到一种魔法般的突破——我们只能期待渐次前行。布鲁克斯的观点难以辩驳，但也难以接受；参加计算机业界会议或是浏览程序员网站时，你总会遇到一些坚称其错的人。

有些人梦想炸毁今天的整座软件大厦，替以某种全新之物。有些人则只一味盼望找到不太顽固、更能响应人类愿望和行为流程的程序员，盼望能得到招之即来、挥之即去的软件，盼望得到足堪依赖的代码。

梦之所寄，行之所为——地狱之门就此洞开。

第 1 章　死定了

[2003 年 7 月]

迈克尔·托伊（Michael Toy）双手托腮，将下巴埋入手腕之间，瞥向他的 PowerBook 电脑，嘴里嘟嘟哝哝："约翰死定了。在下个版本出来前，他得连干五百个钟头……凯蒂死定了。她得干到天荒地老。布莱恩越发要完蛋。而且他只有一半时间了。至于安迪嘛，只有安迪没事。他的工作列表上可没有成百上千的任务。"

其实他们看起来还好。这是一个夏日，程序员们围坐在加州贝尔蒙特（Belmont）的一张普通会议桌旁，倾听着经理的发言。托伊是位高个子男士，有着一副铁石心肠和一条马尾辫，不过看起来他正陷入沮丧之中，历数程序员们是多么地落后于进度。这是 2003 年 7 月 17 日，他感到自己也死定了，只有不到两个月的时间，不可能在下一版本预定完成时间到达之前搞掂一切。

"谁的工作列表不够时间完成，就和我一起重新过一下列表。"

他说的是软件缺陷列表——未解决或"开放的"问题及缺陷。列表

给出了开发者们要修复的所有程序问题。缺陷列表存在于一个叫做 **Bugzilla** 的程序中。托伊手下的程序员们还用 **Bugzilla** 跟踪全部编程任务。完成这些任务，才能发布项目的各个版本；每位程序员都必须将自己的列表输入 **Bugzilla**，并且说明每个任务预计花费的时间。

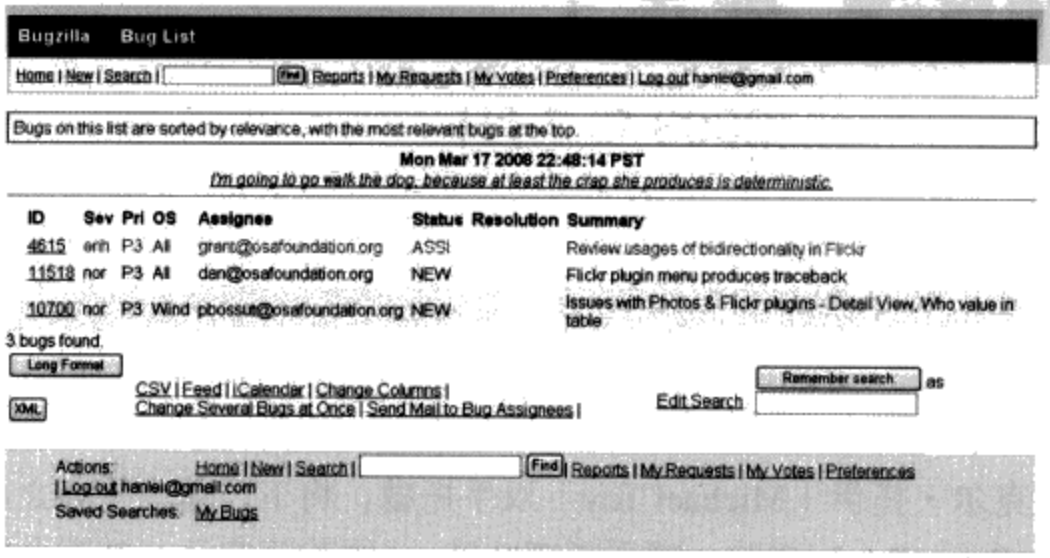


图 1-1　Bugzilla 中的缺陷列表

“不如看看咱们为什么会落后于进度吧。谁先说？”

一时间无人说话。系统架构师约翰·安德森（**John Anderson**）是个编程老手，也是项目的主力程序员。他打破沉默，低声作答：“原因很多。建造之前，先有蓝图。我们一直就没蓝图，所以会碰上没有预料到的问题。除非你确定自己能做出某个东西来，否则很难说会花多长时间。”

“但总不能甩手不干吧。”托伊惯于飞快地查阅日程安排，以“越快结束越好”的态度主持研发会议。但这次他坚持不放过任何问题。“咱们要估算进度，找出可能导致延误的原因。”

项目组中较为年轻的程序员杰德·伯吉斯（**Jed Burgess**）说：“不确定因素太多：你所需的时间取决于其他人所花的时间。”

托伊开始查看安德森的缺陷列表。“臭名昭著的窗体闪烁问题。有进展吗？”

正式而言，这是记录于 Bugzilla 中的第 44 个缺陷，最初于 2003 年 1 月 19 日登记，描述信息是“当修改窗体大小时出现闪烁”。我在 2003 年 2 月的一次会议上听到这个问题，当时开源应用基金会（Open Source Applications Foundation, OSAF）刚完成 Chandler 的一个早期版本，该版本甚至早于 0.1 版，仅在内部使用。OSAF 的程序员由托伊领导。Chandler 的最终目标是成为一种功能强大的“个人信息管理器（Personal Information Manager, PIM）”，用于管理和共享日历、邮件、to-do 列表，还有其他在我们生命中凌乱散落的信息。直到现在，程序仍处于雏形阶段。

在 2 月份的那次会议上，安德森提到过这个闪烁缺陷——在改变 Chandler 软件中某个窗体的尺寸时，屏幕会闪烁一秒钟左右。安德森认为这是个小问题，不过还是应该查实和修正。虽然缺陷不会影响程序运行，但它不符合安德森的审美观。可直到将近六个月之后的今天，他仍然没能修正。

安德森解释说，问题之棘手超乎自己的想象。问题不在自己和同事编写的代码上；出错的根源在于一个称作 wxWidgets 的软件里面，Chandler 小组采用该软件作为项目的构造块。安德森要么等着 wxWidgets 的开发者修正代码，要么就得绕过问题所在。

“开始时以为只要 4 个钟头就可以修复，”托伊说，“看来跟实际情况相差甚远。”安德森平静作答：“这跟寻宝差不多。你要找个入手点。在开工前，要找到线索，而且你不知道花多长时间才能找到。”

“你最初估计的是 4 小时，现在给你 8 小时。”

“有时候，”安德森颇具哲学意味地回应道，“你一早醒来，脑中灵光闪现，于是就手到擒来——大抵如此。”

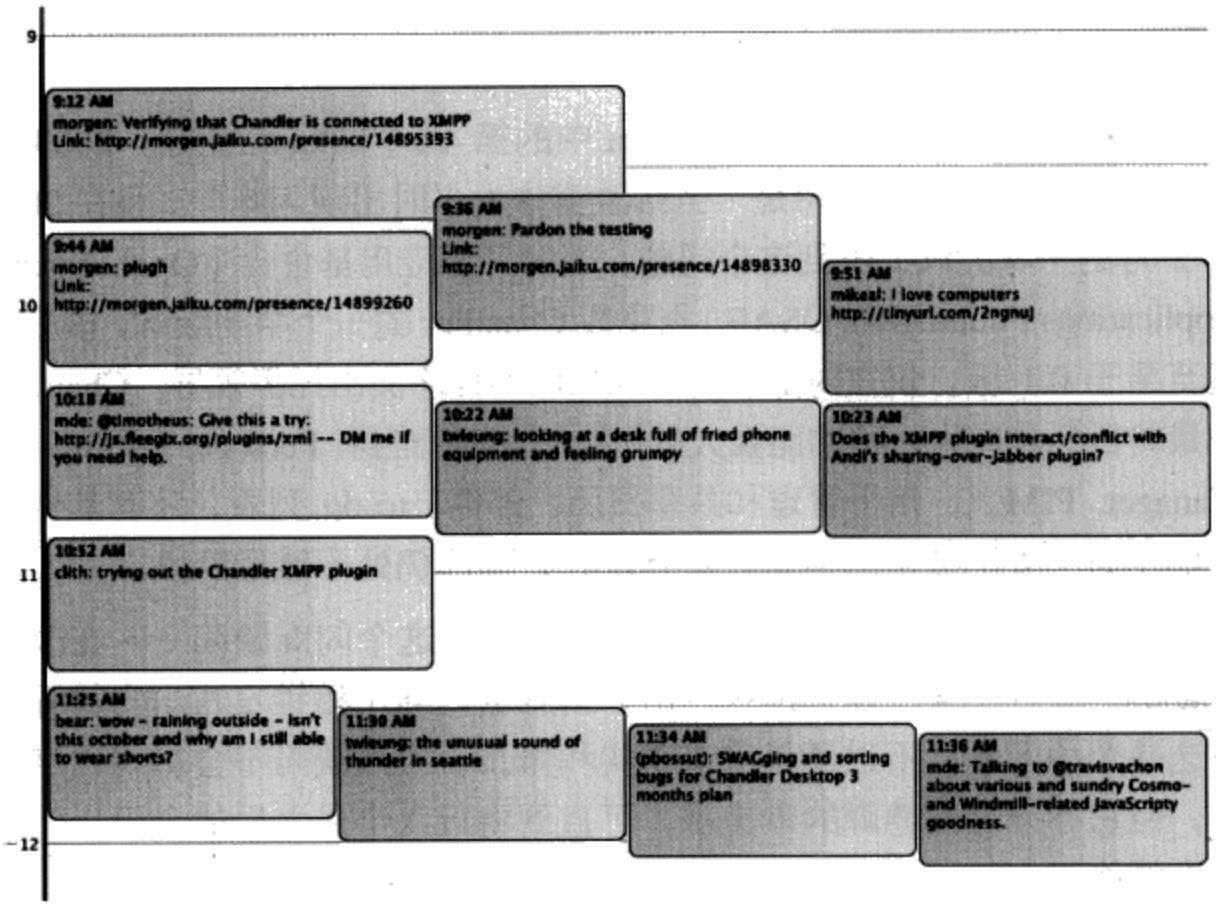


图 1-2　OSAF 开发者工作日历

在他们对话期间，米歇尔·卡普尔（Mitchell Kapor）安静地坐着。卡普尔是开源应用基金会的发起人和资助者，他把 Chandler 看作自己的孩子。他把目光从黑色 Thinkpad 笔记本的屏幕上抬起来，说："是不是可以找出造成这种寻宝问题的原因呢？也许有某类任务导致不确定因素产生？"

"在着手修正缺陷的一个钟头内，"伯吉斯插话说，"你就知道它会变成什么样。"

意见取得一致了：黑洞式的缺陷——即无法确定修正所需时长的缺陷——在 Bugzilla 中应该用特别的警示词标记出来。

开完会没多久，托伊坐到桌前，打开 Bugzilla，给第 44 号缺陷输入一个新关键词。“当修改窗体大小时出现闪烁”：**可怕（scary）**。



使用这种宿命论言辞并非托伊独有的怪癖：面临困境仍不忘幽上一默，是一种程序员文化。托伊在网景公司（Netscape）工作时，开始建立自己的特别词库。如今，人们认为是网景这家网页浏览器公司的软件和股票引发了互联网的爆炸式增长。但是，网景公司的程序员当年却把自己看作被诅咒的军团，项目期限遥不可及，注定将以失败告终。

说句良心话，OSAF 的程序员们不该有此怨念：他们中有几个刚去参加了一个大会，向一帮拥趸演示了 Chandler 项目的进展。有人认为，Chandler 的愿景不太靠谱，不过大多数听众都看着 Chandler 的蓝图说：“最好现在就有得用！”软件工业一蹶不振已达三年之久，这些程序员正为一个由卡普尔投资 500 万美元创建的非盈利组织工作。项目计划虽然野心勃勃，不过这帮人也都是编程老手，取得过令人钦佩的成绩。安迪·赫兹菲尔德（Andy Hertzfeld）编写了早期麦金塔（Macintosh）操作系统的核心代码。约翰·安德森为麦金塔系统编写过最早的字处理软件之一，稍后还在史蒂夫·乔布斯（Steve Jobs）的 NeXT 公司中负责管理软件部门。没有到会的 Chandler 程序员罗·蒙特利（Lou Montulli）则是 Netscape 浏览器关键代码的作者。可在托伊眼里，他们都死定了。

其实第 44 号缺陷并没有什么特别的让人恐惧之处。自从计算机软件从逐行文本之乡迁移到现在的图形化窗口-鼠标之地，缺陷就变成了程序员们必须为之负责的例行公事。让托伊害怕的并非第 44 号缺陷本身，而是无法确知需要多长时间才能修正缺陷。以此类推，历数 Chandler 中类似的不可知因素，加起来就变成了开发经理的噩梦：日程中的“黑洞”，

充满不确定甚至不可知因素的时间陷阱。

两个月前，Chandler 项目的 12 人开发队伍在一周内连续开会，试图解决一系列他们名之以“蛇”的难题。“蛇”这种说法也来自于托伊对网景公司垮台的教训总结。所谓“蛇”，并非只是难题，而是一种“大家没能一致同意如何解决的问题”。在 OSAF，“蛇”取代了“龙”，成为描述此类现象的专用词。

黑洞啦、蛇啦、龙啦——这些隐喻给软件项目管理披上了一层神话英雄的外衣：问题在于，如何安排多个程序员真正完成开发任务。无论是整天不断地往 Bugzilla 里添加条目，还是不停开会推动进度，软件时间始终像是一条大蛇，而且似乎无法征服。

对于在座的 OSAF 程序员来说，这不是什么新概念。软件项目难以按进度安排实现，这种情况极为常见，而且为众人所宽容。在软件开发世界里，进度延误普遍到人们特意生造出一个委婉词来描述它：**slippage（失速）**。

当然，每个领域都存在延误现象。司法诉讼程序蜗行般的进度尽人皆知；建筑承包商要是按期完工，就会招来怀疑的目光。然而，软件时间自我扭曲再头尾相接，如同莫比乌斯环[①]一般令人费解。进度忽而突飞猛进，忽而不知何故驻足道中。在你以为大功即将告成之时，却又山穷水尽，花上整半年时间，一无所得。

感觉像是这样：在软件开发项目中暗藏有一条线缆。线缆系紧时，进展迅速。线缆断开时，工作停滞。局中人上下求索，想找到断点所在；观局者先是试图指点迷津，尔后耐心尽失，猛批一通。

历史上任何软件项目均曾有其隐藏的线缆。对于改进软件开发所做出的努力，都是为了让线缆保持系紧。

[①]德国数学家莫比乌斯发现将一个纸条的一端反转 180 度与另一端对接在一起，就形成了一个奇妙的环，后来人们为了纪念莫比乌斯的这一发现，将这样对接形成的环称之为“莫比乌斯环（Mobius）”。

对软件时间导致问题的最早也是最好的诊断，来自弗里德里克·布鲁克斯于 1975 年撰写的《人月神话（The Mythical Man-Month）》一书。布鲁克斯曾是 IBM 的资深程序经理，他亲眼目睹了 IBM System/360 主机操作系统的打造过程，该系统是当时规模最大的软件项目。那台巨大而昂贵的大型计算机，本应是后来二十年中生意场上的中流砥柱，但它的孕育和诞生却饱受延误和成本超支之苦。正如布鲁克斯所言，它变成了一个“沥青坑”，一个能粘住大企业的陷阱，哪怕对于 IBM 这样“孔武有力”的公司也不例外。布鲁克斯所在的部门，深受进度落后之困，不断招募新程序员参与项目，如同李将军（General Lee）[①]派遣一个又一个旅团进攻墓园山脊（Cemetery Ridge）[②]——结果发现后援力量非但不能挽救项目，而且会让状况变得更糟糕。

布鲁克斯法则就这样产生了，听起来既是原则也是悖论：“往已延误的项目中补充人力，只会使其继续延误。”

布鲁克斯法则在面世后的三十年间声望日隆。但它是如此地违反直觉，常使程序员和开发经理陷入狼狈的境地，他们宁愿装作法则并不适用于自己，也不肯与之妥协。

布鲁克斯写道，软件开发者通常都是乐天派，他们认定每个缺陷都可以被迅速修正，且修正旧缺陷必能减少新缺陷的数量。这种盲目乐观，加上程序员想要取悦主顾（他们委托程序员创造一些新鲜事物），往往让进度在一开始就偏离正轨。布鲁克斯发现，在实际开发中，编码只占软件项目开发时间的 1/6，有一半时间用于测试和修正缺陷。但只有少数项目经理会真正按照这种思路来安排开发人员的工作时间。

稍后，布鲁克斯提到，“在预估及安排项目进度上的每一分努力”都

[①]美国南北战争中南军领袖。

[②]位于弗吉尼亚州亚当斯县葛底斯堡城南山脉，1863 年葛底斯堡战役战场之一。该场战役以南军败退告终，是美国南北战争中死伤最为惨重的战斗。

是“危险且具欺骗性的神话”。所谓“人月”，是一种科学管理概念，它假定生产力可被拆分为不连续、无差异、可替换的单元。所以，如果 100 个人能在一个月内完成 50 个部件，则单个部件需要 2 个人月——故而只要往项目中投入更多工人，就能更快地生产出相同数量的部件。想在 3 天而不是 30 天内生产出 50 个部件么？那就投入 1000 个工人吧。

布鲁克斯观察到，“只有在任务能分派给许多**互相之间无须沟通**的工作者时，人和月才是可互换品。”对于软件而言，项目各有差异、工具不断升级，每当团队中加入一个新组员，老组员就得放下手边的工作，帮助新组员进入角色，每位组员都要等待重新分派任务，好让新组员有事可做。在你意识到这一切之前，已经远远落后于进度了。布鲁克斯发现，在最坏的情况下，这会导致灾难般的延误循环，一种“再生性进度灾难”。每次重新安排进度计划，都导致雇用更多人力，于是又不得不重新安排进度。布鲁克斯对此心存恐惧：“疯狂在兹。”

布鲁克斯发现，制作软件的大量工作受困于“序列约束”，它限制了任务分解的程度：完成某项任务是处理其他任务的先决条件，这与人力投入多少无关。“十月怀胎，”布鲁克斯写道，“无论多少妇女参加都一样。”

最后，布鲁克斯以不同个体程序员生产力的巨大差异来完成对人月神话的反驳——极好的程序员能在规定时间内完成十倍于普通程序员的工作量，而且完成质量也五倍于普通程序员（以速度和效率而论）。若人与人的之间劳动效果如此不同，则衡量标准也无从谈起。



布鲁克斯法则暗示最理想的开发组规模是一个人——无须停下工作与同事沟通的单个开发者。一切均顺序进行，确保项目维持布鲁克斯所谓“概念上完整”的状态：项目中所有环节目标一致，且按计划完美结

合。事实上，孤狼们在软件历史中贡献良多。但是，太多软件渐成巨物，仅凭一人之力无法做成——包括设计精美、易于使用的图形界面，还有复杂、可靠、用于存储大量可信信息的数据结构等等。最初从布鲁克斯时代的操作系统——管理计算机系统资源并使之在用户和程序间构成回路响应的基础代码——开始，逐渐扩展到软件的各个领域，编程演进为一种团队努力，一种团队活动。

按照业界标准，OSAF 的 12 位程序员组成了一个相对较小的开发组。在 2002 年 10 月 Chandler 首次官方声明之后的 9 个月里，他们经历了布鲁克斯在数十年前描画的种种困境。尽管他们采用了诸如邮件列表、blog、缺陷跟踪、源码控制等工具，但和他人保持一致仍然极其困难。每次延误，总让人想要雇用更多人力，但新增的人力似乎根本无助于推动进度。

然而，更要命的是，OSAF 的工作给布鲁克斯出了道新题。卡普尔创办的 OSAF，名字来源于“开放源代码软件”，这种开发方法论在《人月神话》的年代尚未出现。与之后其他任何一种开发方式不同，开源让编程世界错以为布鲁克斯法则将成弃履。



在购买商业程序——或称“塑封软件（shrink-wrapped software）”——时，你会得到一份随时可在电脑上运行的文件。那是一种所谓“二进制”或“机器可执行”文件：它由机器代码组成，满是 0 和 1 的组合序列，大多数人根本无法读懂它，甚至大多数程序员也读不懂。这种二进制文件并非人工的直接产物，而是编译器的产物——编译器是一种计算机程序，它读入程序员用 Java 或 C 或 C++等语言编写的代码，并将其翻译成机器语言。

软件公司售卖二进制文件，因为你需要它来使用他们的产品；他们也喜欢二进制文件，因为它能保护机密。比如，假使你想了解 Microsoft Word 是如何编写的，光看二进制文件没有用；得读到源代码——人类为编译器撰写的成千行代码。但是，和大多数商业软件企业一样，微软不会让你接触源代码。程序源代码是商业软件公司最重要的资产，知识产权法规的柏林墙保护着这些财宝。

这让比尔·盖茨的公司得以创立历史上最赚钱的软件特权。这也意味着如果微软的软件产品出了问题，即便你是个编程老手也无法修补它。你得等着微软来修正问题，因为只有微软程序员才能接触到源代码。

学术计算领域则一直有开放之风：源代码的自由共享，传承了科学研究各领域的自由共享传统。20 世纪 80 年代和 90 年代，微软问鼎个人计算软件产业之际，开源只余星火之光，但并未熄灭。20 世纪 90 年代中期是互联网出现在上百万台个人计算机桌面上的全盛期，这是学术界借助政府资助而培育出的结果。具体而言，学术界开发出一系列协议，让不同公司制造、由不同硬件组成、运行不同操作系统的计算机之间得以互通。将计算的孤岛连接为一个全球网络的软件概念，并非出自追求利润的企业办公室中，而是根源于在大学或公众资助研究中心辛勤工作的理想主义研究者们“发布且共享”的心态。



理查德·斯托曼

在 20 世纪 90 年代，奇客式狂热和协作理想的世界开始自我觉醒。程序员通过他们创建的网络看到对方，眨眼间，意识到他们之间共有一种知行观，与微软支配的个人计算主流全然不同。从理查德·斯托曼（Richard Stallman）和李纳斯·托瓦茨（Linus Torvalds）这两位核心人物处，他们获得了灵感和工具。

1985 年，麻省理工怪才斯托曼出于对商业软件产业封闭代码积习的憎恨，创办了自由软件基金会（Free Software Foundation）。基金会发布了一种特殊的软件许可，依据该许可，你能够获得全部所需代码、复用

这些代码、组合到新产品中——但利用该代码创建出来的任何产物，也必须采用同样的许可。这种被称作 GPL（GNU Public License）的许可显然意在限制将自由程序私有化的行为，但是批判者将它看做是可恶的传染病毒。比尔·盖茨认为 GPL 是反资本的，并嘲笑它是吃豆人（Pac-Man）——电视游戏中胃口超大的怪家伙。不过对于斯托曼及其伙伴而言，GPL 是抵御商业侵蚀的有效路障。



图 1-3　比尔·盖茨嘲笑 GPL 是吃豆人

斯托曼和他的战斗队韬光养晦数年之久，暗中努力搭建 GNU——一个旨在创建占据各大学计算中心的自由 UNIX 操作系统版本的项目。与此同时，有位名叫李纳斯·托瓦茨的芬兰年轻程序员，在互联网新闻组里贴出了有关他自己做的一个小操作系统项目的消息——“只为兴趣，不与 GNU 争短长”，在 1991 年的一篇帖子中，他这样写道。项目吸引了一些志愿贡献者，托瓦茨欢迎他们加入，并把他们组合到现在所谓的 Linux 社群中。（Linux 为 GNU 贡献了原本缺少的中心构件，操作系统“核心[kernel]”，数字大脑的要害。）

20 世纪 90 年代，被纯化论者称为“GNU-Linux”的操作系统，逐渐成长为稳定、免费的 UNIX 版本，并且完全向外界开放。商业世界开始接受它，而创建它的程序员们则着手检视开发方法、总结实践经验，并把归纳出来的模式称为：开放源代码软件开发（open source software

development）。

开源倡导者们喜欢强调“免费啤酒那样的 free”和“自由演讲那样的 free”之间的区别：并非所有开源软件产品都免费，但所有开源软件均可被自由查看、摘选和复用。开源世界内部山头林立：例如，斯托曼的追随者们视“开放源代码”一词为过时货，只垂青于“自由软件”的说法；关于不同许可方式的争论也带有宗教般的狂热。但是，各种意见的鼓吹者都拥有同一信仰：每个人都可修改的软件会随时间的推移得到改进，而这正是“封闭”软件所无法做到的。

媒体普遍将开源描写为微软的挑战者，事实也正是如此。但这种挑战是非直接的。商业软件为赚取利润而生产，很难想象在这方面谁能比微软做得更好。要与微软抗衡，唯有改变游戏目的。设若做软件不为赚钱呢？设若让程序员从中找到乐趣并秀出奇招呢？诚然这听起来像是不符合常理的乌托邦。然而，设若自我驱动的程序员真能写出更好的代码呢？设若他们的效率更高呢？



艾瑞克·瑞蒙德

开源并不仅给出了一种生产和分发软件的替代性经济基础方案；它还能彻底改变软件开发的具体过程——将其从少数隐士手里拿出来，散播到广大人群中。程序员艾瑞克·S·瑞蒙德（Eric S. Raymond）在 1997 年撰写的论文《大教堂与集市（The Cathedral and the Bazaar）》中，道出了对这种改变的最佳阐述。瑞蒙德是一名老派黑客，他描写了自己在意识到托瓦茨技法的意义时所受的启发：

“多年来，我一直在宣扬小工具、快速原型和进化式编程的 UNIX 福音。可我曾相信，存在某种重要的复杂性，需要更集中且有计划的手法

来解决。我也曾相信，最重要的软件……需要像建教堂一般，由独立的巫师或一队相互隔离的魔法师精心打造，在面世之前绝不发布 beta 版本。李纳斯·托瓦茨的开发风格——早发布、多发布、全委托、尽开放——让我吃惊。这里不存在静穆、虔诚的教堂式开发——相反，Linux 社群看似一个乱哄哄的大集市，铺陈了各种日程和手法……要从中得到前后一致和稳定的系统，简直只能指望奇迹再三出现。可事实上这种集市风格看来行之有效，真令人震惊。”

《大教堂与集市》从个人喜好和外部环境等方面分析了托瓦茨在 5 年多时间里管理 Linux 项目开发的方式，并总结出一系列原则。“好的软件开发工作始于打造开发者本人。”只要是做某种取悦自己或满足己需的东西，程序员就会动力十足，努力做到最好。“好程序员懂得写什么。卓越的程序员懂得改写（并复用）什么。”若能取之于现有，程序员决不肯创之于本无。

在 UNIX 世界中，程序员向来习惯于自由分享源代码。但对于重要工作，大教堂模式仍是当时首选，对于自由软件和商业软件均是如此。为 Linux、BSD UNIX 做了最多前期工作的人是比尔·乔伊（Bill Joy），太阳微系统公司（Sun Microsystems）的创始人之一。同样也是在大教堂模式下，斯托曼的自由软件基金会造出了广泛使用的 GNU 软件工具集——比如直至今日仍在许多程序员桌面上运行的 emacs 编辑器等等。托瓦茨和 Linux 改变了现状，证明“杂乱”、集市风格的做法，也可用来制造大型、有用、不断改进的软件。在瑞蒙德撰写《大教堂与集市》时，Linux 和其他开源软件项目——如 Apache Web 服务器（时至今日，已占有三分之二的市场份额，而来自微软的竞争产品则只占到 25%）——的成功已然开始吸引大众的目光。

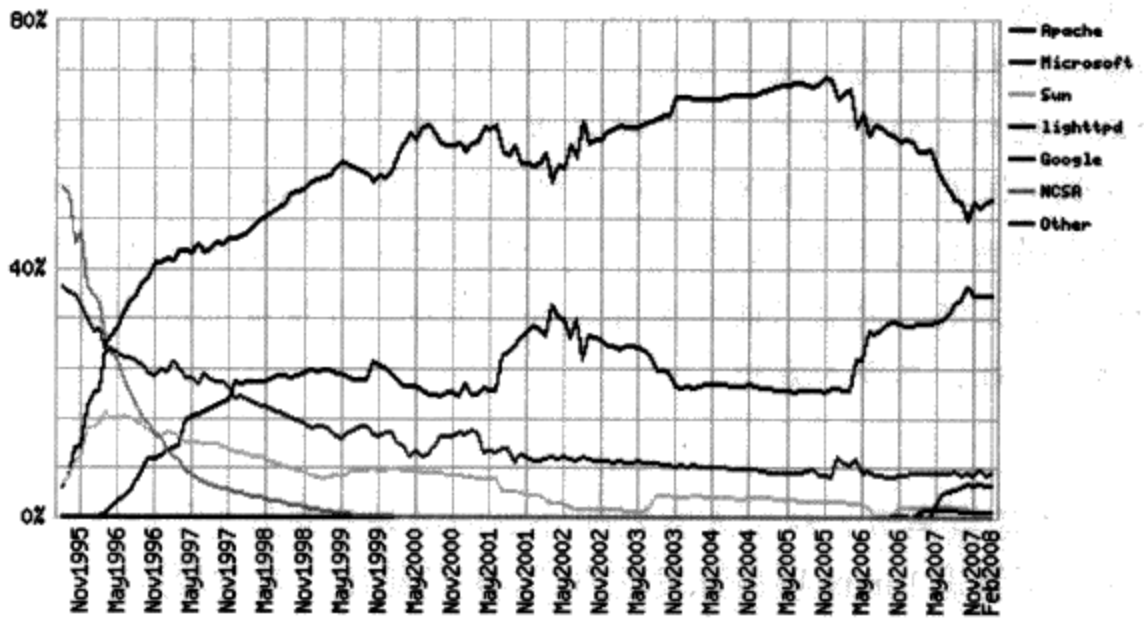


图 1-4 Web 服务器市场占有率

不过瑞蒙德却是从托瓦茨的工作中看到超越布鲁克斯法则的第一人——这是一种新的软件开发模式，可集众程序员之长，且免于让项目身陷泥潭。瑞蒙德指出其中的两个要诀：其一，低成本、广泛地接入像互联网那样的网络，让开发者之间能建立迅速、可信的沟通渠道，存储可被开放访问的共享知识和代码池；其二，围绕一种领导方式——如托瓦茨那样的方式——形成合作团队的良好风气，欢迎新人进入、鼓励成员做出贡献，同时尽可能增加合格成员。

瑞蒙德写道，一旦具备这些条件，就能彻底改进在程序中寻找和修正缺陷的过程。与其害怕竞争对手窃取代码而秘不示人，不如将其展现给全世界，邀请同道加入。而且，假使你独具魅力，且懂得如何管理一群开发者，或许还能组织起自循环的集智体系。

“只要有足够多的 beta 版测试人员和开发者队伍，几乎所有问题都会很快被发现，而且总有人知道该怎么修复。或者用不太正式的说法，‘眼球足够多，缺陷无处躲。’我把这叫做‘李纳斯法则’。”瑞蒙德声称，这

种由网络助力的开放式单点复查新方式颠覆了弗里德里克·布鲁克斯的残酷悖论。“兹对布鲁克斯法则提出以下不同意见：让懂得采用怀柔政策的开发协调人员拥有至少像互联网那样好的沟通媒介，多人必好过一人。”

瑞蒙德的论文在编程界引起了一波兴奋的浪潮。让布鲁克斯焦头烂额的 IBM 大型机项目正好符合瑞蒙德关于“大教堂编程模式”的描述。在 20 世纪 90 年代后期，大教堂模式仍被采用，华盛顿州雷蒙德市微软的巫师们尤为如此。而在城墙以外，成片帐篷中传出的喧闹之声已无法充耳不闻。

从大教堂的僵局走向开源集市的高效，这条路并不像开源鼓吹者们说的那样通行无阻。瑞蒙德也承认，布鲁克斯本人至少预示了推动新运动的几条原则：在《人月神话》中，他写道，“对于一个广泛采用的程序，维护费用往往高达开发费用的 40%甚至更多。令人意外的是，这部分成本受用户量影响相当大。用户越多，发现的缺陷也越多。”实际上，在布鲁克斯的年轻时代，程序源代码通常可以自由获得：当时软件还没有成为一个价值数十亿美金的产业的基础；它只是计算机生产厂商为了使计算机能够运转和有用而提供给用户的东西，并且可以随意共享。1975 年，正值个人计算机软件产业和微软公司即将诞生之际，布鲁克斯在他的书中定义了一种完整的“编程产品”，这种产品“每个人都可以使用、修正和拓展”——听起来比较像是李纳斯·托瓦茨而非比尔·盖茨干的事。布鲁克斯还抱怨有些程序“未完整交付（没有源代码或测试用例）”。布鲁克斯认为，完整的程序应包括源代码，以备后用。

开源本身，即让程序员学习和扩展源代码的可能性，已经是布鲁克斯描述的世界之一部分。但是，开源方法论和基于互联网的协作集市，仍然待以来者。瑞蒙德之洞见，更多地体现于看到互联网和托瓦茨式的领导方式在让接触源代码更具有价值方面的重要性，而不在于解释为什么应该让程序员接触源代码。

《大教堂与集市》令人信服地阐述了托瓦茨式的开放源代码是一种飞跃，但并未完全解决布鲁克斯所看到的、在从头搭建一套新软件的过程中遇到的困难。瑞蒙德展示了开放集市如何能够融大群程序员之智慧于一炉、且不触及布鲁克斯法则的铁律，但他却无法说明开源方式是否更易预测开发新程序所需时间，或者是否能加速软件面世的进程。《大教堂与集市》并未真正驳倒布鲁克斯法则，也没有解决软件开发中的时间难题；它描绘了另一个编程世界，在那儿，时间无关大局，因为志愿者们因为乐趣和自我而协同工作，不求物质回报。

“乐享即能高效，”瑞蒙德写道。“愉悦是金。”这些理想化的格言令人振奋——但需经实践验证。开放源代码似乎也受困于延误。自瑞蒙德的论文发表以来，风头最劲的开源项目是 **Mozilla**——诞生于网景公司废墟之中的网页浏览器。在从与微软的商战中最终败退后，网景公司转向开源，公布了浏览器的源代码。结果造就了今天的 **Firefox**，一个拥有数百万用户的强健且继续改进的自由程序。**Firefox** 的确已经开始侵蚀微软 **Internet Explorer** 占据的市场。而在 **Mozilla** 旷日持久慢慢成长的同时，微软也巩固了其优势地位。开源运动的新集市模式在很多方面改变了计算世界，但说到能比大教堂模式更快地让新产品面世，却并无显著建树。

而在 **2003** 年夏季，如何让新产品面世正是卡普尔及 **Chandler** 开发者们心中之痛。项目组开发模式混杂：在运作模式上像是一个开源项目，他们在互联网上公布源代码，致力构建让志愿开发者加入的社区；然而他们又像是传统的软件创业公司，有一组核心程序员努力在限期内做出新产品。卡普尔是莲花开发公司（**Lotus Development Corporation**[①]）的创始人，后来又做过数年的风险投资，创业公司模式是他所熟知的，而开放源代码则是摸索前行的方向。

卡普尔创建 **Chandler** 的动力来自于对 **PIM** 软件的反思，他野心勃勃想要做出一种能运行于三种主流个人计算机上、让用户方便地共享数据

[①]莲花开发公司于 1995 年以 35 亿美金的高价被 IBM 收购。

的 PIM 软件。正应了瑞蒙德的名言，“挠自己的痒处”。卡普尔渴求这样一种软件，而现有的产品与他心中的图景不甚契合。开源独行侠可能会马上开始往互联网上张贴一些代码，但卡普尔没有这样做。他先转租了一套办公室，这正是企业家的做法。

和许多初创软件公司一样，OSAF 最早的办公地点也是因陋就简。卡普尔在硅谷靠近旧金山半岛边缘的贝尔蒙特找了一处地方，办公室所在的建筑是一幢不太抢眼的低层写字楼，与企业级数据库大厂甲骨文公司（Oracle）总部的玻璃大楼只隔一条路。“Oracle 大学”也在这座办公楼里。后来旁边开了一间 Extended Stay America 家居式旅馆的分店，接待到甲骨文公司总部出差的客人。但在 2002 年春天，这所写字楼还空空荡荡的。卡普尔的新创公司搬进了四楼角落的办公区，原租户是一家叫做 Reactivity 的软件公司。这就是硅谷不断自我更新的方式，就像一片森林：技术工业如植物般周期性地生长然后枯萎，在前一轮生长期中诞生的公司消失后，新公司就在它们留下的空地上萌芽。



图 1-5　占据整面墙的白板

OSAF 的开发者们地尽其用，在一面墙那么长的白板上涂涂写写。白板写满后，又往彩色玻璃上写。我 2003 年 1 月第一次去拜访时，他们看起来像是办公室耕地里的农夫，在显示器的营火前凑成一团，想要发

动一场革命。

2003 年 7 月，OSAF 从贝尔蒙特小屋搬到了旧金山城里一座厂房式建筑的顶层。这座建筑被完全改造为符合环保要求——可循环木制品、太阳能供暖等等，而且屋顶上还有遛狗场，符合卡普尔“环境对狗友好”的需求。许多开发者都养狗，卡普尔夫妇也在前一年刚领养了一对拉布拉多狮子狗，取名为 Cosmo 和 Chandler。（卡普尔用项目名给狗取名。）

新办公室很漂亮，天花板上横亘着多条屋梁。不过，和其他新的或不怎么用的办公室一样，也有点像是一处遗址。有人在白板上写了几个大字，“Welcome to Howard Street（欢迎来到沃德街）”。跟着，有人擦掉几个字母，再加上一个字母，这句话变成了“We come to ward three（我们来到三号牢房）”。后来有人觉得这句话太凄怆，就擦掉了字母“r”，留下较为积极的圣经式预言“We come toward thee（吾等行向你处）”。



图 1-6　沃德街的新办公室

卡普尔多次告诉程序员们，非营利组织 OSAF 不能像那些在互联网泡沫时期由风险投资支持的初创公司一样运作。那些公司纷纷关门大吉，在旧金山湾区遗害不浅——OSAF 所在街道旁边的市场大道（Market Street）南端，就像一片死域。在 5 月的一次会议上，卡普尔安抚一位害怕开发组发展得过快过大的成员，说道，“咱们不玩硅谷那种死亡竞赛，只求按期交付产品获利，把产品质量抛诸脑后。有类似经历的人——在座的绝大多数都经历过——都知道是怎么回事。重蹈覆辙，只会血本无归。”

卡普尔没有食言：OSAF 稳步成长，到夏末，许多程序员甚至开始担心新人加入速度太慢。理论上，采用开源模式意味着要让大量志愿者来推动 Chandler 项目前行，而且的确有许多程序员定期登录到 OSAF 的 IRC 聊天频道，关注 Chandler 的代码发布。不过，外部贡献者没有很多事可做。“我们只是在玩开源，”我听到过不止一位 Chandler 工作人员这样说。当时是 2003 年，项目似乎毫无进展，人们度日如年。不管是投资者要求的发布期限还是大量的开源代码，都无法让 Chandler 从牢狱中脱身。



米奇·卡普尔——莲花开发公司的创始人

六月份，迈克尔·托伊开设了 blog，讲述他作为 Chandler 开发经理的工作状况。在第一篇帖子中，他反思了为什么 Chandler 的进度如此慢如龟行。迈克尔比较了他在 OSAF 和网景公司的工作经历，把部分原因归咎于 OSAF 更为民主化，缺少等级式结构——无论是大教堂还是集市都认为，等级式结构是将顽固的程序员组织起来的有效手段：“在 OSAF，我们遵从集体意见，很慢才能做出决定。最重要的是做出好决定。在网景公司，写代码是重中之重，所以只要是有助于解放人力、有助于开始写代码的决定就是好决定。OSAF 的问题在于，真的很难为编写代码扫清障碍。”

托伊还感受到另外的不同之处，那就是 Chandler 是个新项目。其他许多开源项目都是重写；Chandler 则野心勃勃重起炉灶。“在网景公司，浏览器已

是成品，只需重写即可。在 OSAF，我们采用全新的特别手段存取 Chandler 数据。每个决定中都包含了不确定性，地面还没凝固，怎么能站得稳？”

弗里德里克·布鲁克斯在三十年前观察到，“构建软件最难之处在于决定说什么，而不是怎么说。”OSAF 证明这句话仍然正确。

托伊在 blog 上袒露心声后六个星期，我和安迪·赫兹菲尔德谈了一下。在 OSAF 会议上，赫兹菲尔德总是坚持让开发者们停止设计、开始编码——至少不要等到地面完全凝固才开始。赫兹菲尔德告诉我，“我的风格是赶快干起来，然后再把它变成我们想要做的大东西。这不是个平庸之作，是个大东西。不过总得开始干吧！要点在于激情开干——然后顺其自然。船到桥头自然直。”

托伊将问题归结为：“做大事没有大压力也可以吗？或者，压力是天才不可或缺的一部分吗？”



除了关于 Chandler 的思索，托伊还在 blog 上抒发对棒球运动的热情，甚至贴出他在 8 月某个周四下午错过的一场巨人队比赛的照片。如果托伊带本《人月神话》去太平洋贝尔公园（Pac Bell Park[①]）看球，在局间休息时翻翻，可能会看到这样一段话：“棒球队经理视进取精神为伟大球员和伟大球队的天赋要素。奔跑快过所需，移位先过所需，努力大过所需。对于伟大的编程团队，这也是必备要素。”

[①]旧金山巨人队主场

埋头工作 9 个月后，产出的幻想多于代码，Chandler 项目程序员们仍未找到进取方向。

“愉悦是金，”艾瑞克·瑞蒙德写道。“开源的成功告诉我们，对于创造性工作，玩耍是最经济有效的模式。”

此话说来容易。秋天将至，Chandler 项目组中仍毫无愉悦可言。

第 2 章 Agenda 之魂

[1968 年～2001 年]

周四早晨，Chandler 开发组的大约一打成员齐聚在位于旧金山沃德街 543 号 5 楼的大会议室，召开每周工作例会。笔记本电脑连着 WiFi 无线网络；咖啡杯冒着热气；狗们四处晃荡，人们互相交谈。突然，米奇·卡普尔从两扇门之间探出头来说，“来了个你们认得的人。”他带进来一位身穿蓝色外套、看起来很眼熟的高个子男士。这男士站在那儿，静待说话声平息下去。

访客说话了。“我是阿尔·戈尔（Al Gore）[①]，差点儿就成了美利坚合众国的下届总统。”除了那些狗，大家的下巴都惊愕得掉了下来。

前副总统是来参加卡普尔主持的一个社会责任投资基金会议的。看来他不太明白怎么会站到一群程序员面前，而且也没有太多话好讲，只说了一句“继续努力！你们正在改变世界！”然后就转身走出去了。

不太清楚戈尔对 Chandler 和 OSAF 有多少了解。不过，把宝押在说他们正在“改变世界”上，是明智之举。这个短语在硅谷由来已久。二

[①] 美国前副总统，2000 年竞选总统失利。

十年前，史蒂夫·乔布斯（Steve Jobs）游说时任百事可乐首席执行官的约翰·史考利（John Sculley）加入苹果公司（Apple），问后者是愿意毕生“卖甜汽水”呢，还是愿意过来制造电脑、改变世界。

听得多了，就会觉得这话天真得让人难以置信。不过，Chandler 项目组成员们——和所有其他软件开发者一样——都抱着自己的工作能改变人们生活的期望投入开发。或许，这种大多数程序员所共有的理想主义直接源自编程的辛劳和挫败。与令人畏惧的抽象建模较劲，或者面对蜂拥而至的缺陷大军，唯有万丈雄心能助你艰难前行。

戈尔可能搞错了时态——Chandler 项目并没有真的“正在”改变世界（至少尚未开始）——但他直切要害。Chandler 项目正是为改变世界之梦所驱动。不过，项目开始时谨小慎微，带有些许怒气、些许心中之痒。

01010101010101010110101101110

话说 2001 年，旧金山城区米奇·卡普尔办公室外的机房里，放置着一台 Microsoft Exchange 服务器。米奇压根不想进那间房，可是在这新千年伊始他经常不定时地出现在那儿，因为他是 Exchange 出问题时的第一求助对象。卡普尔的妻子弗丽达·卡普尔·克莱恩（Freada Kapor Klein）当时在同一办公地点运作一个小型非盈利基金会，基金会成员们使用微软产品来共享日历并管理邮件。

Exchange 是个大程序，足以应付规模数百倍于这个基金会的大公司的数据流管理；它是许多“微软商店”——基于比尔·盖茨的产品构建自己软件的公司——的核心。卡普尔对盖茨及其立场的憎恶有据可查。他曾被冠以“反盖茨”的名号。他不乐意使用 Exchange，但当其时——2000 年和 2001 年，硅谷技术创业潮起又潮落——小型组织的日程管理

没有其他方案可选。Exchange 太过强大，功能远超小机构所需。Exchange 也很昂贵：得置办一台服务器、购买 Windows 许可、购买 Exchange 软件许可，如果没有全职技术人员，还得雇个咨询师，请他每月来几个小时做系统调整。在如梦方醒之前，你已经为保持日历同步花费了上千美金。

应该另有坦途可行。为什么非让微软插一脚？何不让你计算机上的日历与同事的计算机直接沟通？采用“端到端（peer-to-peer, P2P）”方式，网络中单台计算机之间就能互相直接连通。服务器只是个中间人；中间人可以被踢出局。

拜 Napster 一时风光所赐，P2P 成了在线世界的热门话题。Napster 使用 P2P 技术，让数百万用户得以交换音乐文件。音乐行业和法院横加干涉，关闭了 Napster，但从那时开始，P2P 就流行起来了。首先，P2P 行之有效。其次，不论计算机与用户数量怎么增加都能承受，而且不需要花钱购买中心服务器设备。最后，它响应了那种推动早期个人计算机革命的反独裁诉求：推翻计算机牧师——那些在数十年前运行大型主机、现在又护卫着服务器机房的大牛们——让每个人都能叫桌上那个机箱盒子绽放异彩。

那就是卡普尔自 20 年前开始，作为一名个人计算先锋所走过的路。如今，他知道自己还想做软件，而且对要做什么也考虑了很久。也许，就做这个。



20 世纪 80 年代初期是个人计算机爆炸性增长的年代。1982 年，《时代（Time）》杂志弃名人而选计算机，在年末特刊封面上印出“MACHINE OF THE YEAR”的大字。对于关注过那时风潮的人来说，卡普尔之名

和一个叫做 Lotus 1-2-3 的程序永远地连接在了一起。Lotus 1-2-3 并非电子表格的首创；电子表格始于丹·布瑞克林（Dan Bricklin）的 VisiCalc，这个程序让商业世界看到，比上一代柜式小型机内存少、运算能力低的桌面微型计算机也能派上用场。而 Lotus 1-2-3 则展示了非凡的灵活性和不俗的设计。VisiCalc 多运行于当时流行的 Apple II 计算机，1-2-3 则专为新型 IBM PC 设计。那时，IBM 进军桌面计算机市场的新品在多数办公室中尚属罕有，但卡普尔大胆押宝，用 1-2-3 软件展示了 IBM PC 的计算能力和运算速度。软件大受欢迎，人们纷纷购买 IBM PC，只为能运行 Lotus 软件。1-2-3 正是计算机工业所谓“杀手级应用”的化身。

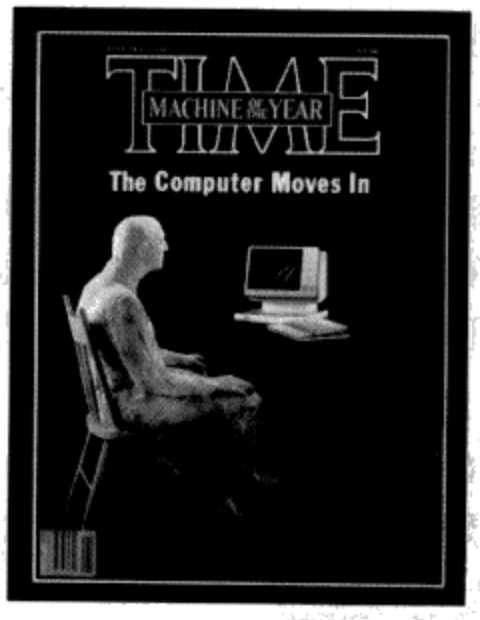


图 2-1 MACHINE OF THE YEAR

当卡普尔用“莲花（Lotus）”这个来自佛教[①]的词给公司命名时，自然不会想到“杀手”之类说法。1-2-3 让莲花公司一夜成名——在 20 世纪 80 年代，它比微软还要大——也让卡普尔陡然暴富（根据《商业周刊（Business Week）》报导，获利约 1 亿美金）。不过他并不开心。1990 年，莲花公司的传奇尚未被遗忘，卡普尔在接受戴维·甘斯（David Gans）

[①]佛教各宗派均极力称颂莲花（莲华），陈译《摄大乘论释》卷十五，以莲华之香、净、柔软、可爱四德，喻法界真如之常乐我净四德。又如以法华、华严比喻法门；《华严经》、《梵网经》等有莲华藏世界之说；密教亦以八叶莲华为胎藏界曼荼罗之中台；以莲华表示众生本有之心莲。

的采访时说："在变成数字资本家之前，我曾教人超觉静坐[①]，还在一家社区医院的精神科做过心理咨询师，这些经历对我影响极深。我拥有心理咨询的硕士学位。所以，我另有志趣。我只是误入计算机领域，无意成为比尔·盖茨——只有比尔·盖茨才能做比尔·盖茨。我向来不求做大公司、赚大钱。我只是办了家叫做莲花的小公司，做了个几百万人争相购买的软件产品，结果这家小公司陡然暴长，员工数千，每年收入数亿美元。很不爽。至少对我个人来说，很不爽。所以我离开了。在某一天，我离开了。"

卡普尔于 1986 年 7 月从莲花公司离职——如他对 Inc.杂志所言，"功成身退"。但还有个项目让他滞留了一段时间。这个项目致力于采用计算机人工智能来管理记录信息的小纸片——名片、随手贴、笔记页——卡普尔那些夏威夷式衬衫的胸袋里塞满了这类纸片。不是所有人都有此烦恼，但生活中的确到处都是信息。计算机在记忆细节方面远胜于人类。个人计算机能否帮助我们管理这些信息呢？买个日历或者地址簿倒是容易，但那些忙人们每天要对付的不便分类的偶发信息又该怎么办呢？

莲花公司有个叫做 Agenda 的项目，正是为了解决卡普尔的小纸片问题而设立。计算机长于以预先定义好的分类存储信息，能很好地应付存货、账本、电话目录之类的信息。卡普尔口袋里的纸片——提醒、要做的事情、好主意、建议等等——很难分类。他想要把这些东西一股脑扔进计算机，过后慢慢分拣。而且，要能方便快捷地找到它们才行。

莲花公司于 1988 年发布了 Agenda 软件。这是个简单的列表管理软件，但有几个突出特性让它跻身于软件传奇之列。你可以用它来管理日常生活，也可以用它来组织学术研究、音乐收藏或是工作项目，需要管理大量细节信息的地方都用得上。其"自动分派"特性——在意义模糊的短语里面，如"下周五与约翰共进午餐"，找出"下周五"这个日子——如同魔法一般；没有其他软件能与之媲美。而且，它还引入了一种管理

[①] Transcendental Meditation (TM)，印度人 Maharishi Mahesh Yogi 创建的一种瑜伽静坐术。

数据的新手段——介于传统计算机数据库的严格结构和字处理软件的自由格式之间。

Agenda 突破了计算机的严格逻辑与人类的语焉不详之间的阻隔，在这片荒土上拓出一片新天地。它的创建者们（卡普尔、后来创办了手写计算[①]公司 Go 的杰瑞·卡普兰[Jerry Kaplan]以及爱德华·J·毕洛夫[Edward J. Belove]）认定这样一些超乎常规的原则：用户不用关心软件的存储结构，只管输入数据就好；用户应该能够容易地扩展和修改数据结构、添加新分类，且不会导致数据丢失；用户应该能够用自己创建的新方式查看数据——也可以在自己创建的视图中操作和修改数据。

[①] Pen-computing，手写计算/输入。Go 公司造出了 tablet computer 的第一个原型。

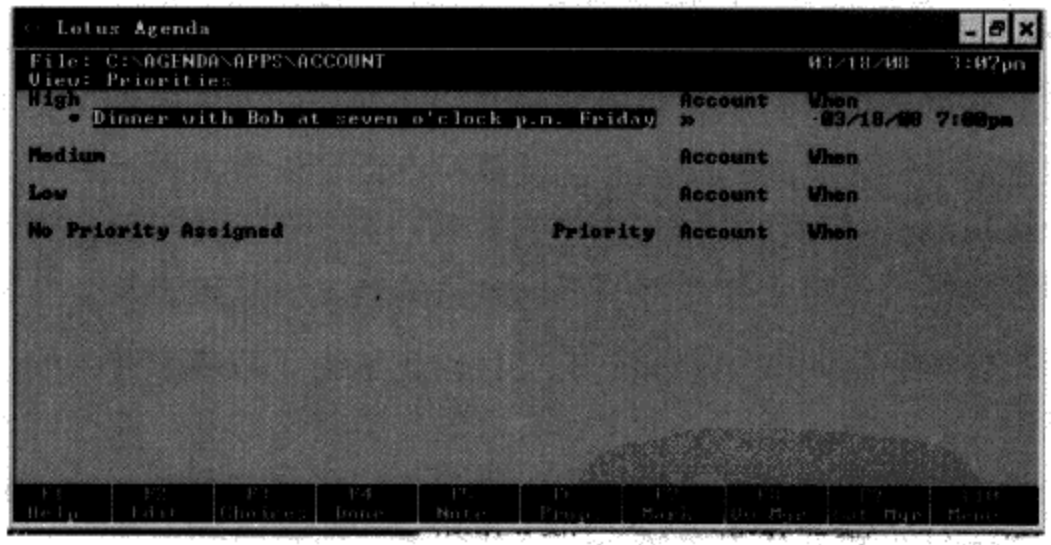


图 2-2 Agenda 能接受灵活的输入格式

看看二十年后的今天，我们使用的软件中也只有少数能做到上述几点，你就能想象设定这些目标是多么胆大妄为。程序或网站总是要你按**它设定的**方式而不是**你自己的**方式填空——社会保险号码里面不得包括连字符！信用卡号中不得包括空格！——而 Agenda 早有独门秘技让用户随意输入。

Agenda 推出后好评如潮，吸引了众多追捧者。赞美之声至今充斥网

络。1992 年詹姆斯·菲罗斯（James Fallows）在《大西洋月刊（The Atlantic）》上撰文称："在我用过的所有计算机程序中，Agenda 绝对是最有趣的，也是两、三种最有价值的软件之一。"然而，万千宠爱集于一身的 Agenda 却没能达到"杀手级应用"的境界。部分原因是已经成为巨兽级企业的莲花公司急于在商业市场大获全胜，不知道该怎么利用这个软件，也可能是作为创始人的卡普尔弃公司而去，让莲花公司无心销售 Agenda 软件。还有，Agenda 运行于微软为 PC 机打造的原始操作系统 DOS 上，而未来显然是麦金塔和 Windows 那种图形操作系统的天下。（卡普尔说，莲花公司开发了 Agenda 的 Windows 版本，不过没有发布。）为了描述 Agenda 开拓的新软件类别，莲花公司市场部门发明了一个词组"个人信息管理器（Personal Information Manager）"，这个词组的缩写形式如今仍在使用：Microsoft Outlook 就被看作是一种 PIM。随着 20 世纪 90 年代的来临，借用软件业中形容不再受厂商支持的程序的说法，Agenda 被莲花公司"遗弃"了。

或许这个软件太超前于时代。或许它是错了位的公司战略的无辜牺牲品。又或许正是由于其创新性和灵活性对大多数用户而言功能太多——如菲罗斯所言，"太强大了，简直不是给人用的。"



卡普尔离开莲花公司后，在麻省理工大学教了一段时间的软件设计。他创办了一家名为 On Technology 的公司，做计算机网络办公协同软件。他投资创立了电子先锋基金会（Electronic Frontier Foundation），一个在线的公民自由权利组织。在互联网的早期发展过程中，他做了一些富有远见的投资，得到投资的公司后来成为 20 世纪 90 年代互联网高峰期的弄潮儿。当多数专家们仍热衷于类似美国在线（America Online）、

CompuServe、Prodigy 之类封闭网络的“墙内花园”时，他就投身于开创开放网络的工作。稍后，他离开波士顿来到硅谷，在 Accel 合伙人公司（Accel Partners）做了一名风险投资家——很快他就发现自己不适合这份工作。（“我总向着企业家，”后来他跟我说，“而作为风险投资家，很多时候必须反其道而行之。你得遂少数几个合伙人[主要投资者]的愿，他们有时志趣相投，有时同床异梦。”）

不过他始终放不下 Agenda。他说自己一直“和这个产品骨肉相连，就像是自家孩子一样”。他渐渐从风险投资事业中抽身，雇用了几个程序员，使用 Java 语言构造 Agenda 的新版本。不过结果不尽如人意。“感觉小里小气，显得老土，”卡普尔说。“就像是制造第一辆 T 型福特车[①]。现在我们懂得好多关于汽车的事情啦！”他珍视的并非 Agenda 的“特性列表”——软件的种种特殊功能——而是动态适应性的程序精髓，即“先扔进去，延后处理”。在考察 Microsoft Exchange 的不足之处、开始梦想发明它的替代品时，卡普尔决定，无论新软件外观如何，都必须具备 Agenda 之魂。

当 Chandler 项目开始萌芽之时，卡普尔唯一确定的就是，这个新产品应该开放源代码。2001 年春，互联网新经济壁断垣颓，旧式的营利型软件创业公司也不甚符合市场口味。在电子邮件和个人信息管理市场上，普遍认为与 Microsoft Exchange 和 Outlook 争锋没有前途。但卡普尔选择开源却并非单纯的投机行为。他顺应了 20 世纪 90 年代后期的开源运动潮流；他发现开源概念令人信服、深合己意。

对于那些靠投资软件企业发家的人，或是那些从塑封包装程序中掘得第一桶金的人来说，开源不会是第一选择。回到 20 世纪 80 年代仍在莲花公司的那段日子，有一天，卡普尔颓然呆坐在自己的办公室中。窗外，自由软件运动的启蒙者理查德·斯托曼，那个不修边幅且脾气糟糕的家伙，正领着一批来自“编程自由联盟（League for Programming

[①]Model-T Ford，福特公司 1908 年-1927 年间的车型。在制造该车型时，首次在密歇根工厂引入了“生产流水线”的工艺。福特公司也凭借这种耐用的低价车，一举坐上汽车工业头把交椅。

Freedom）”的卫道士们在街上抗议。他们反对莲花公司的软件专利政策，抗议那场旨在禁止程序员模仿 Lotus 软件用户界面特性的“look-and-feel 官司[①]”。卡普尔其实同意抗议者的观点，不过他虽然已经不是 CEO，但也有责任支持自己创办的公司；正是诸如此类“不可忍受的个人冲突”导致卡普尔索性一走了之。

现在这些冲突都没有了。卡普尔成了一位细心的慈善家，开源让他得以在从事其所热爱的软件设计的同时，感到自己也在为公众作贡献。“我和李纳斯·托瓦茨等人吃午饭，”卡普尔回忆道，“确信那并非疯狂之举。”开源运动推崇的非传统概念——你是说无人拥有代码？将代码四处散布？——正与他独立不羁的理想主义相契合。托瓦茨面向结果的开源案例，也符合卡普尔作为商人所信奉的实用主义。

开源运动曾在一支纷乱舞曲中与商业世界共舞数回。2000 年春，股市对互联网概念的疯狂追捧达到顶点，Linux 世界也吹起了小小肥皂泡，包括红帽（Red Hat）和 VA Linux 在内的一些开源公司上市了。随后网络股泡沫骤灭，开源概念股和其他技术投资一样遭遇困境。不过，公司机构还是需要运行计算机系统。继 2000 年股市崩盘后，9/11 事件更让各公司机构紧缩开支，此时开源系统的价格就显得吸引力十足了。

刚开始接受 Linux 及类似系统时，反对开源的呼声不绝于耳。批评者认为，如果代码不属于某家公司，就会得不到技术支持。其实另有许多公司提供开源系统的服务，而且越来越多程序员和系统管理员精通开源代码。批评者还说开源项目总会“分叉”——技术上的分歧导致另立山头。没人独有代码，所以任谁都可以引领一个新方向。这样就会出现一些名字相似的分支产品，让人不知所措；不过，既然有经验丰富的开发者持续改进核心项目代码，还有什么好担心的呢？最后，批评者说，开源方法并未证明自己能够开发出让远离计算机科技的人群也能用上的产品。

实情大抵如此，此言一语中的。卡普尔希望他的新项目能予以回击。

[①]Look-and-feel，所见即所感界面技术。当时，包括苹果和莲花在内的一些大公司，对竞争者提起诉讼，要求法庭禁止被告继续在产品中采用和自己软件类似的界面。微软当时是应诉者之一。如果原告胜诉，今天也许不会存在 Microsoft Office、Windows 等软件。那次大诉讼也催生了编程自由联盟（League for Programming Freedom）。

托瓦茨被看作是 Linux 操作系统的“仁君[①]”，他对开源运动一向持冷静乐观的态度。在 2004 年《商业周刊》的一次访问中，他说：“在科学领域里，人们互相查看、引用各自的成果，整个系统建立于这个基础上。而在魔法界则有人暗藏秘技，也不会让别人真正理解乃至使用。传统软件就像是魔法。历史上，魔法最终消亡了。历史将在软件开发中重演。当问题趋于严重，就不能允许个人或个别公司保有秘密。应该让所有人共享知识。”

2001 年春，米奇·卡普尔成立了非盈利的开源应用基金会，招聘了第一批雇员。他后来说，当时需要一些“有双巧手”的人做创意工作，所以雇用了两位年轻程序员，摩根·萨奇（Morgen Sagen）和艾尔·乔（Al Cho），他们曾同在熄了火的网络公司 Excite@Home 中做事。

当时，关于自己要做的软件，米奇只知道三个要素：它应当开源；它应当挠到 Exchange 的痒处；它应当承继 Agenda 之精髓。除此之外，他别无所知。

使用计算机管理信息大潮的梦想，自计算机发明之初就有了。在 1945 年一篇题为“《像我们一样思考（As We May Think）》”的论文中，美国政府二战研究项目领导者范纳尔·布什（Vannevar Bush[②]）提到了 Memex 的设想。他构想，Memex 是一台内置磁带录音机的台式机器，能为研究人员保存资料。布什的 Memex 以其创见开拓了计算机的全新领域。在之后的几十年里，它鼓舞着发明家们不断努力创造新技术，扩展人类大脑的能力。

[①]原文为 Benevolent Dictator for Life。Benevolent Dictator 一词源自柏拉图的《理想国》（The Republic），指怀有仁厚之心、引导社会向好的方向发展的统治者。Eric Raymond 在《大教堂与集市》中，把 Torvalds 称作 Benevolent Dictator for Life，意指 Linux“由少数人掌握核心”的开发模式。

[②]范纳尔·布什作为罗斯福总统的科学顾问，领导了“曼哈顿计划”，制造出世界上第一颗原子弹。后又领导或参加了氢弹、登月飞行、星球大战计划等国防项目。同时，他的一系列著述和教学工作，影响了许多计算机/互联网界先锋，本人也被誉为“信息时代教父”。

在 Memex 之梦的追随者中，鼠标之父道格拉斯·恩格巴特（Douglas Engelbart）当属最有影响力的一位。恩格巴特曾做过雷达工程师，后师从诺波特·维纳（Norbert Wiener[①]）学习控制论。1950 年的一天，当恩格巴特从睡梦中醒来，脑海中灵感突现：世间问题多多，纷繁复杂，人类必须更快变得更聪明，方能尽在把握。他发愿尽余生之力来做一个“增长人类智慧的框架”。从 20 世纪 60 年代初期起，受斯坦福研究所（Stanford Research Institute）的支持，他召集了一班研究人员，开始突破概念和技术的限制。1968 年，在旧金山会展中心，他做了一次传奇般的公开演示。该次活动的视频资料仍可在网上找到，谁都可以点击网上的链接，观看恩格巴特向全世界介绍链接、鼠标，以及其他一些今天被广泛采用的元素。

看一下这些视频，你就能了解到，恩格巴特的“oNLine System（在线系统，NLS）”就是一种 PIM。NLS 的目标是让用户能够“保存并研究想法，使之相互关联，并能交叉引用”。现在的计算机用户大概会把这叫做 outliner（大纲工具）——以可折叠和展开的节点对信息行进行层级结构化组织的程序。但 NLS 可以在网络上共享，而不仅限于单机使用——如恩格巴特在 1968 年向那群极受震撼的观众们所展示的那样，在旧金山会展中心和 30 英里外门罗公园（Menlo Park）的 SRI 办公室之间共享数据。今天看来，NLS 不断闪烁的单色屏幕和粗糙模糊的字体已是老古董，但其功能和设计却树立了协同软件的标杆，现代系统克服重重困难才得以企及。



图 2-3　道格拉斯·恩格巴特的鼠标

[①]美国数学家，控制论创始人。维纳在第二次世界大战期间从事雷达和防空火力控制的研究，以后研究通信和控制技术，于 1948 年发表了著名的《控制论》一书，用统一的数学观点讨论了通信、计算机和人类的思维活动，提出了自动化工厂、机器人、由数字电子计算机控制的装配线等新概念，促进了通信、计算机、机器人和人工智能等理论的发展。

恩格巴特在 1968 年的演示中，让观众看到用 NLS 创建、保存和共享一个食品单有多容易。但 NLS 的真实目的却在于让恩格巴特的程序员们能更好地编程。在 1962 年关于增进人类智慧研究计划的论文中，恩格巴特阐述了程序员最有可能成为初期目标用户的原因。他给出了九条理由，包括程序员对计算机的熟悉程度、面临要解决问题的智慧挑战等等。但他也提到，“成果也可用于智慧增进研究项目自身，改进研究和开发智慧增进系统编程活动的效能。设计、实现和修改程序的能力，在衡量研究进展时颇为重要。”换言之，若 NLS 能帮助程序员更好地编程，则程序员就能更快地改进 NLS。这就出现了正向循环。用恩格巴特的话说，这就是“提靴带（bootstrapping）”。

在恩格巴特看来，提靴带（bootstrapping）的意思是“让改进的过程得到改进”。今天，bootstrapping 这个词容易让人联想到计算机启动叫做“boot up”。拉着靴带把自己提起来是不可能的事，早期计算机系统的建造者们借用这个概念来形容开机和运行的矛盾。初次启动计算机时，内存是空的。这就造成了鸡与蛋的悖论：计算机硬件需要操作系统软件来装载程序——包括操作系统本身。计算机系统发明者们通过一个叫做“bootstrap loader（靴带装载者，引导程序）”的小程序逃离了困境。这个小程序让机器具备刚好足够把大操作系统装入内存的能力，开始正常运行。在早期计算时代，人类操作员可以使用机器上的开关按钮或纸带手工输入引导程序；现代计算机把它存储在内存芯片内。

恩格巴特认为，就达到“智慧增进”程序的目标而言，提靴带更像是一种抽象、一种奥秘，而不仅是一个工程上的难题。提靴带关乎“研究成果的正向反馈，用于改进研究者自己赖以继续工作的手段”。这是一种“三阶现象[①]”，它并不改进过程——如让人更快地解决问题。它改进的，是改进过程的速率——如怎样才能快速教会他人更快地解决问题。

这并非唾手可得之事，也是恩格巴特的项目并没像鼠标那样取得巨

[①]非线性光学概念，利用相关的光学特性，能得到特定波长的激光，或实现频率转换、调谐等。

大成功的原因之一。另一个原因可能是他太执着于保持“智慧增进”项目版本的纯洁性。计算机发明领域的后来者均称颂“可用性”之重要，而恩格巴特则对软件的简单易学没抱太多指望。计算机是用来延展人类智慧的，恩格巴特希望它强大而多用；他可不愿意削足适履，满足人们在几天或几个星期内学会使用的需要。办公室文员惯于使用熟悉的打字机键盘，但恩格巴特相信，他发明的“和音键盘（chord keyset）”让使用者效率倍增，值得花时间去学习掌握。所谓“和音键盘”，看起来就像是 5 个钢琴按键的组合，熟手可以用它单手输入文字。他梦想人类和机器能“共同进化”：机器改变人，增强人们的工作能力，反过来人也持续改进机器。当研究者们聚集到斯坦福的实验室、将 NLS 织入自己的生活时，的确可以看到类似的情况发生：恩格巴特的传记作者希尔瑞·巴蒂尼（Thirerry Bardini）写道：“惊讶的访客报道说，[恩格巴特的研究组开发出]一些奇怪的密码和暗语，比如，他们能用一种怪异的标记语言互相沟通，有些成员向房间另一头的同伴打手势，手指在空中比划出键盘上某个特别的和音键位。”

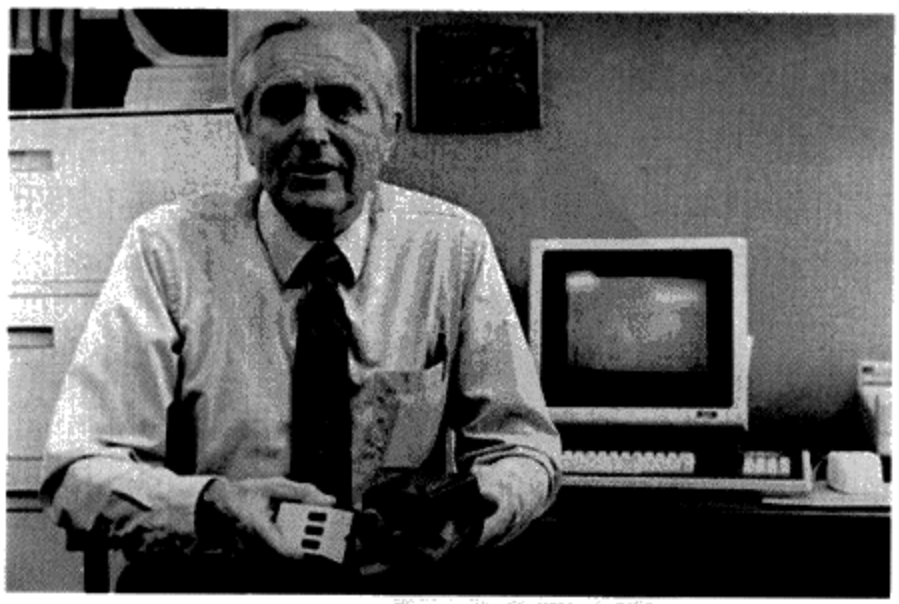


图 2-4　道格拉斯·恩格巴特介绍和音键盘及鼠标

在 1968 年那段录像中，你可以感受到那种怪异：恩格巴特头戴耳机，一手拿鼠标，一手敲和音键盘，看起来像是一位地心旅行导游员，坚信自己将把团队带向光辉的未来。然而，自 1968 年那次演示之后，项目就陷入了混乱。恩格巴特计划持续改进 NLS，而很多年轻工程师则想要另起炉灶、跟上逐年升级的更强大的硬件。慢慢地，团队忘记了自己的使命。20 世纪 70 年代中期，在人类潜能运动[①]（Human Potential Movement）和维纳·艾哈德（Werner Erhard）[②]的 est[③]课程冲击之下，项目搁浅了。

恩格巴特关于人类适应机器的想法，在后来几十年里，得到了一些人的响应。如计算机先驱艾伦·凯（Alan Kay）[④]所言，“不论好坏，恩格巴特都是在致力于打造一把小提琴，”——但是，“大多数人并不想学拉小提琴。”简易与强大、方便与精细之间的拉锯战，贯穿了后来软件发展史的每一阶段。关于可用性原则，我们中的多数人一开始都有一种可以理解的成见：计算机应该让工作变得更容易；**为什么不**让它们干重活呢？但是，将恩格巴特的程序斥为“对抗人类”并不公平，因为其全部目的就是找出如何让技术帮助大幅提高人类思考效率的方法。

计算机科学家杰隆·拉尼尔（Jaron Lanier）[⑤]说过一件轶事。某次，年轻的恩格巴特遇到麻省理工大学的马尔文·明斯基（Marvin Minsky）[⑥]，人工智能领域开创者之一。明斯基吹嘘说，他的研究项目将大大增强计算机的思考能力。恩格巴特冷言相对：“那些都是为计算机做的——你打算为人类做点什么呢？”



米奇·卡普尔总把恩格巴特引为自己的灵感之源，而 Agenda 在某种意义上则是 NLS 的传承者。虽然 Agenda 没有 NLS 那么大的野心，而且也因为对 DOS 的依赖而受到削弱，但它对有意学习新技巧的人类用户

[①] 兴起于 20 世纪 60 年代初期的心理学派。该学派认为，大多数人都有未被开发的卓越潜能。

[②] 本名 John Paul Rosenberg，1935 年出生于美国宾夕法尼亚州费城。后把联邦德国当时的经济与劳动部长 Ludwig Wilhelm Erhard（后任总理）和量子力学创始人之一 Werner Heisenberg 的名字拼到一起，改名为 Werner Erhard。Werner Erhard 是 est 的创始人。

[③] 艾哈德综合“现代心理健康科学”、“心灵动力（Mind Dynamics）”、Silva Mind Control、禅宗思想结合，成为一套研讨课程，称为 Erhard Seminar Training，简称 est。est 曾在美国流行，反对的人也不少，因为有人参加过之后，精神崩溃甚至自杀。引起很多诉讼。est 后来改名为 the Forum，最后停办。

做出了同样的承诺，承诺让他们获得新的能力。在卡普尔及第一批雇员开始描画新项目的轮廓时，在充斥各种可能性的未来之海上，这种承袭自 NLS 的精神引导他们前行。

继恩格巴特的研究项目后，数十年间，计算领域的革命层出不穷。计算机从办公室里走出来，进入千家万户，摆到每个人的桌面上。恩格巴特的一些旧同事，和别人一起，在施乐的帕罗·阿尔托（Palo Alto）研究中心开发出图形用户界面（Graphical User Interface, GUI）。随后，苹果公司的麦金塔计算机让 GUI 得到普及。互联网把全世界的计算机连接起来，组成一片巨大而不受约束的全球性公有区域。

无人可以抹煞这些新事物改变了世界的事实。另一方面，也很难说它们已经像恩格巴特期望的那样，让人类的智慧得到爆炸式的增长。每种提升人类生产力、方便性或创造力的计算机发明，看来都有其负面影响；系统崩溃、病毒、蠕虫侵袭着我们的进步之梦。那么多激情洋溢的鼓吹者为我们描绘出数字化进步的美妙图景，可他们对程序员努力把脆弱代码锻造成型的痛苦记录却只字不提。那记录由一个接一个的灾难组成，在计算领域的历史时间线上，留下了累累弹坑。

今天，任何打算开创一个大型软件开发项目的人都得与这种极度令人气馁的历史包袱相抗争。它让胸怀大志的新来者遭受重创，好像是在对他们说，**你凭什么认为自己与众不同**？



2005 年 1 月 14 日的《纽约时报（New York Times）》重点报道了华盛顿特区发生的麻烦事："由于遇到技术和计划上的问题，联邦调查局在一项旨在提高反恐能力的计算机项目上投入的 1 亿 7 千万美元可能打了水漂。"FBI 投资 4 亿美元，花费 10 年时间，开展名为 Trilogy 的计算机

④第一个完全面向对象语言 Smalltalk 的发明人，2003 年获图灵奖。艾伦·凯曾经是专业的爵士吉他手，在他眼中，易用性永远是计算机科学家和开发者必须牢记的原则。

⑤VPL 公司创始人，"虚拟现实（Virtual Reality）"概念的提出者。

⑥1969 年图灵奖获得者。他搭建了世界上第一个神经元网络模拟器 SNARC。

现代化项目，打算替换掉 3 万台桌面计算机，搭建更为安全和现代化的数字通讯网络。在进行到耗资 1 亿 7 千万美元的第三个模块——一个叫做 Virtual Case File 的软件，它连接新系统，让全国的 FBI 特工随时可以获得最新数据，J·埃德加·胡佛（J. Edgar Hoover）[1]的徒子徒孙们从此走出纸面时代——时，FBI 撞上了冰山。最初 Virtual Case File 目标有限，只是要为 FBI 基于大型机的古董级文档存储系统添加一些网页式的用户界面。后来，Virtual Case File 慢慢膨胀成主要项目，计划创建一种“全新的协作环境，可用于搜集、共享、分析证据和情报数据”（见商业杂志《信息世界[InfoWorld]》）。9/11 事件后，承包方发现 FBI 受到事件的刺激，将需求列表陡然拉长。FBI 期望在 2003 年 12 月前得到成品，但到期时只拿到一个缺陷缠身的原型版本。承包方抱怨说：难道 FBI 不明白，如果他们“变更需求”、扩展了项目，最初设定的交付日期就会不切实际？或许 FBI 当时也经历了计划不断变化的痛苦，因为它不断地更换 CIO。

[1]前美国联邦调查局局长，曾居此位达 49 年。

“慢动作式的出轨，”参议员帕崔克·利亚（Patrick Leahy）这样形容 Virtual Case File 项目，就好像是在国会委员会的房间地板上四处散落着项目残片一般。内部人士把 Trilogy 的名字戏改为“Tragedy（悲剧）”。这种局面对于有关各方应该并不稀奇。FBI 遭遇的困局，在公共软件领域，有一千零一个灾难故事交相呼应。美国国内税务局（Internal Revenue Service，IRS）与 FBI 同病相怜，甚至有过之而无不及：IRS 在过去 40 年里，3 次尝试改造计算机系统，至今一无所成，仍主要依赖于 20 世纪 60 年代用一些数字设备拼凑起来、修修补补的大型机系统。20 世纪 70 年代，卡特总统叫停了第一次 IRS 系统升级；第二次是在 1995 年，白费十年劳力和 20 亿美金后，国会取消了那次失败的升级项目。最近一次努力则受挫于严重的延误和预算超支。人们将失败原因归咎于一些常见问题：对需求列表的不断更改或添加新项；IRS 消极短视；CIO 频频更换；预算和进度安排不切实际。

在政府软件项目年鉴上，记载了无数宗类似的案例，其中包括 30 年前的导弹防御系统，还有正在进行的五角大楼“未来战役系统（Future Combat System）”计划——该计划基于复杂的新硬件和尚处于设计阶段的软件系统（“雾件”），旨在用“信息中心”模型重建美国军队。在美国之外，状况也好不到哪去。例如，2004 年 11 月，在被《卫报（Guardian）》形容为“政府历史上最严重的计算机崩溃事故”中，英国养老金系统全面中止运作。事故的起因，不过是在将 7 台运行 Windows 2000 的计算机升级为 Windows XP 时，不小心把升级范围扩展到数千台还没准备好的处于工作状态的计算机上。

可别武断地认为政府是罪魁祸首；私有工业亦同此凉热。企业界中也充斥了灾难故事，例如麦当劳（McDonald's）的“Innovate”项目——耗资 1 亿 7 千万美金，计划把整个快餐链升级为“实时网络”，让管理层能随时查看每一锅炸薯条的精确状态，项目以失败告终。还有福特公司的“Everest”采购系统，成了个耗资数亿美金、历时 5 年的黑洞。把大量巨无霸项目以失败告终的原因归咎于公司的赘疣和异常以及政府官僚作风的肆意疯长，好像顺理成章。然而，在各种软件项目中，无论大小、无论公私、无论新旧，都可以看到类似的悲惨故事。撇开具体细节不谈，模式令人郁闷地一致：标靶移来移去。目的忽上忽下。计划不切实际。期限一拖再拖。预算膨胀超支。绝望已极。混乱不堪。



1995 年，马萨诸塞州一家叫做斯坦迪什集团（Standish Group）的咨询公司发布了一份关于软件项目失败的研究报告，这份报告的标题是《CHAOS Report（混乱报告）》。该公司总把 chaos（混乱）大写，像是什么缩写词一样，其实用的是原义。斯坦迪什公司调查了分布于大、中、小型公司的 365 位信息技术经理。最后发现，只有 16%的项目获得成功

（“按时并在预算范围内完成所有计划的功能特性”）。其余项目中，有31%“被削减”或被取消——完全失败。剩下的53%被委婉地形容为“待解决项目”，其实就是预算超支、延误以及/或者不能交付计划中的全部功能特性。

你可以批驳该项研究的设计和目的，但不能说它只给软件产业带来了坏消息。斯坦迪什公司继1995年的报告之后定期发布更新，显示出这个领域正在稳步而缓慢提升项目成功率——2004年的报告显示，相比1995年，数字好看了一些，有29%的成功率和18%的失败率。但“待解决项目”则维持在53%的水平上。尽管有所进步，但仍有三分之二以上的项目不能交付。

回到IT产业喜用的关于做软件和建桥梁之间的类比，1995年那份报告认为，软件的问题不只与中途纠正和后期设计更改有关，这些情况都是桥梁建筑师所不能忍受的；它还存在无法从失败中吸取教训的问题：“如果桥塌了，就要做调查、写报告，说明失败的原因。而在计算机产业中，失败案例总是被遮盖、被忽视并且/或者被说成是合理的。结果，我们不断重复同样的错误。”

实际上，有整整一架子书是写软件灾难的，如《软件开发的滑铁卢（Software Runaways）》[①]和《死亡之旅（Death March）》[②]。这些著作开列了一个又一个厄运缠身的失败项目。这些关于技术雄心搁浅在人类愚蠢之滩上的故事，构成了一种虚构甚于事实的文本。故事总是这样的：一群激情洋溢的技术工作者开拔去解决一些奇特而棘手的难题；他们被坏经理、不断变化的需求以及自折兵马的行为所背叛；只好留待会计师和律师们收拾残局。

这类书中最新的一本是笔下凌乱但富有激情的《软件极限（The Limits of Software）》[③]，作者是工程师罗伯特·布瑞切尔（Robert Britcher）。布瑞切尔经历过一场空前的软件大灾难：联邦航空管理局（Federal

[①] Software Runaways: Monumental Software Disasters, Robert L. Glass 著，电子工业出版社有中文版

[②] Death March, Edward Yourdon 著，电子工业出版社有中文版

[③] The Limits of Software: People, Projects, and Perspectives, Robert N. Britcher 著

Aviation Administration，FAA）的先进自动系统（Advanced Automation System, AAS）。这套系统旨在对航空管理系统进行现代化改造；它从1981 年启动，在花费了几十亿美元后于 1994 年宣告终止，颗粒无收。

IBM 于 20 世纪 60 年代为 FAA 搭建航空控制系统时，布瑞切尔就是开发团队中的一员，之后他又被派到项目的替换计划中。布瑞切尔在书中写道，AAS“可能是有组织工作史上最严重的崩溃事件”。FAA 给新系统立了几条规矩：每个比特的代码都要求能被复用。整个系统必须是“分布式”的，即每个空管员的工作站都要和其他工作站实时同步信息。新系统必须在现有系统保持运行的情况下无缝切入。（“就像是给一辆正在收费公路上行驶的汽车换引擎。”）纸张打印或许还管用——在旧系统出问题的时候，空管员就得依靠这些东西——不过 FAA 否定了这个方案：它太老套。如此等等。

在高峰时期，AAS 项目每天要花掉政府 1 百万美金；同时有 2000 名 IBM 雇员为该项目工作，给每行代码写上 100 页文档。但到最后，布瑞切尔下了个结论，“这套软件不可能写得出来。”FAA 的需求远远超出了人类和机器的工作能力。“若你想感受一下，”他写道，“可以读读《伊利亚特》。”《软件极限》本身就是一本由被诅咒的桌面程序战士撰写的《伊利亚特》，充满了能人们被更强大的力量所困的悲剧宿命论。在布瑞切尔的书中，项目成员们被挫败感而不是工作负担压垮，有人砸烂自己的汽车，有人发疯，有人自杀身亡。一位项目经理吃纸上瘾，随着进度一再延误，在开会时他往自己胃里塞的纸片越来越大。无人能够全身而退，包括该书作者在内。

有个我认识的工程师这样形容 AAS：你住在一所还算舒适的房子里，发现冰箱可能有问题。冰块有时会融化，门也关不严实。修理工差不多个把月就得过来修一次。你注意到，这冰箱既笨重又费电，你看上了希尔斯百货公司（Sears）里面陈列的那些新款冰箱。是时候了。你当

即出发，先到几个州看地块，考虑建一所新房。你请到贝聿铭和另外几位大建筑师，做了个初步设计。做这个花了点时间，所以你还得继续忍受那台旧冰箱，它发出烦人的噪音，在夜里叫人难以成眠。你看了几套方案，甚至建造了一、两个原型。时间流逝，你终于选定设计方案。开始建造前，举办了一场狂欢宴。开工。建屋。庆祝继续进行，每块砖都令人激动。然后你改了主意。你其实是想要一所铺着红木地板、有花园的日式房屋。你改造已建好的部分。搬开一些砖块，铺上草皮。新房看起来真不错。某天晚上，你在新屋床上躺着，没听到冰箱发出的噪音。出问题了。安静令你无法入睡。花了太多钱！你压根就不想搬家！孩子们也不喜欢住到新房里。你女儿还说："我恨它。"你只好改弦更张，不搬了。15 年过去了，还花了几十亿美元，那老冰箱仍然运转。大抵如此。



如果程序员太过在意过往那些软件灾难留下的教训，就一行代码也写不下去。每次失败都如此相似，简直令人遍体生寒。你唯有交叉十指，祈祷我们的老冰箱不出毛病。然而，除了那些经历过类似 FAA 案例的老手程序员之外，多数人在做新软件时，都乐于忘记过去、盲信未来，坚信这次会不同。

"程序员都是乐天派，"弗里德里克·布鲁克斯在 1975 年写道，"或许那多挫折把别人都吓跑了，只剩下那些习惯性地关注最终目标的人。或许只不过因为计算机还年轻，程序员还年轻，而年轻人总是那么乐观。"但是，程序员与生俱来的乐天情绪，往往被他们径直杀向困难与问题的行动所掩盖。记者都知道，如果从销售人员或市场人员口中挖不出什么料，不妨去找个工程师问问，他（如果没被隔离在外或被命令闭嘴）会讲出困窘的真相。在 Chandler 的草创期，我和项目首任软件架构师约

翰·安德森初次见面，他坐在我斜对面，没等我开口问，他就说开了："我们遭遇了三次大失败。"假使程序员是天生的乐天派，他们仍独具慧眼，能洞悉坏的一方面。对灾难场景的想象是一种专业素养；为了练习技艺，他们得深思熟虑、巨细靡遗。

意大利激进分子安东尼奥·葛兰西（Antonio Gramsci）[①]曾有名言云，要想改变世界，你得"理智上悲观，意志上乐观"。现今做软件的人大概都没能承袭这种二分法的头脑。

在刚开始设想后来成为 Chandler 的项目时，米奇·卡普尔自然也是个乐天派。当时软件产业正值严冬，跌入谷底。但他是成功的企业家，财力雄厚，能遂己所愿卷起袖子开干，满足自己的、也是世界的所需。

我第一次接触卡普尔并提出要写这个项目的时候，他告诉我，把 Chandler 当作案例研究对象可能不是好主意。"我们不够典型，"他说，"我们开源，我们不营利，我们依据《软件设计宣言（Software Design Manifesto）》（卡普尔于 1990 年撰写的论文）设计优先的原则做软件。我们是在尝试自己鼓吹的东西。"

从某种意义上说，他说对了。Chandler 缺乏一般软件项目中的种种常规约束。没有公司的组织架构去承诺期限；没有缺乏耐心的投资者作壁上观，一旦代码不能发布就拉闸断电。但在我看来，所有这些都让项目更具价值。在观察 Chandler 项目的进展时，我们能剔除金钱方面的影响；财政压力常被看做是软件灾难的罪魁祸首，而 Chandler 则不会受到财政因素的影响。Chandler 可以让我们从技术、文化和心理的角度观察到，在消除了商业世界的苛求后，软件是怎样制作的。假使编程在本质上设有路障，阻碍我们做出软件，也许我们也能在 Chandler 项目上窥其一斑。假设卡普尔及其团队富有远见卓识，能够跨越这些路障，那就更好了。

[①]意大利共产党领袖，奠定了意大利马克思主义文艺理论基础。

在说到项目不够典型时，卡普尔也是在重弹项目领导者们为了忘记过去而发出的老调：**这次会不同**。他把乐观情绪直接扔到了软件产业史和个人史的脸上。卡普尔离开莲花公司、完成 Agenda 软件后，开办了 On Technology 公司。这家公司并没显出有什么不同。1987 年，卡普尔满怀让 PC 软件更聪明的雄心创办 On Technology。他想把 Agenda 识别日期和人名的特性加以扩展，做一个更广泛、开放的产品，运行于所有种类的计算机上，让办公一族能分组协作。虽然没人搞得清楚 On Technology 具体要做什么，但那肯定是令人振奋的产品。然而，一年之后，什么也没做出来。“我们提出了计划，最终又把它枪毙掉了，”卡普尔对《华尔街日报（Wall Street Journal）》的记者说。

卡普尔解雇了一半员工，删减目标，让小团队来执行新计划。两年后，On Technology 发布了首个产品：麦金塔操作系统的一种插件，让用户更方便地组织和查找文件。后来该公司又发布了两个软件产品——其中一个是在计算机网络上组织会议的工具，另一个是协同编辑共享备忘录的工具。这些产品直击需求要害，不过都没能火起来。它们都只能在麦金塔电脑上运行，而这个市场越来越窄、越来越孤立。On Technology 最后也成了软件灾难史上的一页。

卡普尔说对了：Chandler 项目**不够**典型。不过哪个项目够典型呢？

“没有所谓典型的软件项目，”安迪·赫兹菲尔德喜欢这样说，“每个项目都自有其不同之处”。

第 3 章 原型与 Python

[2001 年～2002 年 11 月]

你如何管理音乐收藏？如果不想放任如山的 CD 胡乱堆积，就得想个办法了：也许用音乐家名字的字母排序。或者，分类放置——摇滚放一处，爵士放一处，古典放一处——再在每一类中用音乐家名字的首字母排序。再或者你打算更随意一些，则也可以用 CD 的标题字母、厂牌、发行年代排序。按照购买时间放置也是个办法，就像地层结构是按年代分层一般。

但是，一旦选定整理手段，往后就要一直沿用这种方法了；换用其他方法所需的劳力投入太大，超出了多数人愿意接受的范围。（2004 年，有位旧金山概念艺术家打算按照书脊颜色重新排列一家二手书店里面的全部货品。他形容那次工作需要“20 个人通宵工作，还得保证可乐和比萨饼供应充足”。）这就是现实世界物品的限制。不过那也有好处。只要你习惯了自己的整理方式，就会发现，可以凭借感官记忆找到东西：你的手指记得埃尔维斯·科斯特洛（Elvis Costello）[1]的专辑放在 CD 架的右上部分，因为它们多次在那儿摸到过这些专辑。

[1]著名歌星，朋克音乐新浪潮（New Wave）的代表人物。

开始用计算机管理音乐收藏后，多了一大堆可能性：只要心有所想，就能在转瞬之间重新随意排列。音乐软件让我们能够从前所未有的视角查看收藏，例如“这首歌之前听过多少次？”我们从物理世界抽取一些要素，将其模型化为数据，一切尽在把握。不过，我们也失去了物理世界中存在的简单的线索——在潜意识中引导手指的 CD 盒脊颜色，或者在 CD 机上面放个碟盒，提醒自己下回听这张碟。

把生活的某些方面融入到软件代码中之后，就很容易被各种新奇诱人的可能性所迷惑，看不到自己放弃了什么。设计良好的程序在提供大多数新特性的同时，并不试图对抗进化过程留给我们的物理世界倾向。（我从昨天就记得这个按钮在哪儿，因为大脑能记忆空间中的位置——所以明天最好还能在同一个地方找到它！）

然而，我们常常停用一些软件，这些软件不但令人困惑地与我们能够感知的物理世界相脱离，而且也缺乏“数字化”带来的灵活多样性。想按类别整理音乐收藏吗？好——不过请别指望在程序预置类别之外再添加新类别。原来的文件分类模板适合老鹰乐队（Eagles）和阿姆（Eminem）的音乐，而你是个歌剧迷，需要另外的分类模板？对不起，我们将“音乐家”和“曲目”分类牢牢地钉进了软件结构中，你得搞个新程序，才能按作曲者、演唱者和唱腔分类。

之所以有这种事，并非因为程序员故意给你使绊儿，而是因为他们自己受到工具和材料的限制。软件是抽象的，看似可以无限扩展。但是，就灵活性而言，它却非常顽固，令人发指地棘手，而且常以其强硬一面示人。

在每个编程项目的早期阶段，当团队选择语言和技术、准备切入开发时，这种两难悖论就降临了。对语言和技术的选择，初时无关大局、重选亦可，转而变得混凝土般坚如铁石、关系重大。



2001 年春，米奇·卡普尔、摩根·萨奇和艾尔·乔定期开会，讨论他们要做的软件。萨奇在为 Excite@Home 工作的最后一年中，公司开始分裂，他出于自己的需要和兴趣，做了一个便于查找、分类书签和其他个人偶发信息的 Web 程序。他谈了自己的经验，而卡普尔也描述了从 Agenda 中吸取的经验，他们开始问自己一个看似简单的问题：我们如何组织信息？如何对这种信息组织法建模——需要怎样的数据结构才能让计算机也能回答这个问题？

讨论不断深入；白板已被写满。

软件的单个新功能发布出来让所有人用上，需要很长的时间，所以程序员常常通过所谓“用例（use case）”来测试自己的设想。用例是一种假想场景，模拟用户需要什么、想怎么用软件。OSAF 项目的较早用例之一，是模拟管理庞大的个人音乐收藏，这个用例后来又不断被提出。这顺理成章：在少年漫游时代，卡普尔曾经做过 DJ。而在 2001 年，苹果公司的 iTunes 程序刚刚推出，只能在 Mac 系统上运行，并没有现在这么流行；而 iTunes 的配套设备 iPod 则尚未面世。

萨奇和乔开始草拟数据结构，使之符合上述用例和其他用例的需要。需要哪些类别和标签？这些类别如何互相关联？基础数据（如音乐文件）如何关联到能描述音乐的所有“元数据”，如音乐家名字、唱片标题和曲目数量？如何高效地存储这些数据？以后是不是还有办法往里添加新的元数据？

过了好几个星期，写满好几块白板后，终于有了答案。可以用一种简单的三段式模型描述任何事物，看起来就像英语简单句中主语-动词-宾语的结构：

<this> <has-relationship-with> <that>
（<这个><关联到><那个>）

然后他们发现，10 多年前发明万维网（World Wide Web）的科学家提姆·伯纳斯-李（Tim Bernes-Lee）带领的研究人员，已经描绘出这个答案的轮廓，而且至少是部分地实现了解决方案。伯纳斯-李有一个梦想，他将其称为语义网（Semantic Web）——现有 Web 的升级版，依赖于更聪明、更复杂的数据表现形式。语义网基于一种名为 RDF（Resource Description Framework，资源描述框架）的技术。RDF 以“三元组（triples）”——包括三个部分的语句——存储所有信息，描述事物之间的关系。这与萨奇自己琢磨出来的结构非常近似，而且经过数年的研究之后，已经开始投入细节编定阶段。

“好程序员懂得写什么，”艾瑞克·瑞蒙德在《大教堂与集市》中写道，“而卓越的程序员知道改写（并复用）什么。”没必要重新发明 RDF 轮子——只要搭上顺风车就行了。



音乐库用例在纸上没有停留多久；下一个加入 OSAF 的程序员实现了它。安迪·赫兹菲尔德是在 1984 年创建苹果麦金塔电脑首个操作系统的核心人物。麦金塔电脑引领了之后 20 年内个人电脑的前行之路，设定了硅谷每个新项目至今仍渴望达到的创新和精致的标准。安迪·赫兹菲尔德也因此而名扬圈内。麦金塔电脑发布后不久，赫兹菲尔德就离开了苹果公司，将自己的一手技艺、满腔热情和梦想投入到两个极具野心的项目中，但均以失败告终——第一个是 General Magic，一家致力于革新手持“数字助理”市场的创业公司；第二个是 Eazel，一家还未完成自己的 Linux 程序就不幸遭遇网络股泡沫破灭的开放源代码公司。

赫兹菲尔德早已是开源福音的狂热传道者；在 Eazel 关门大吉的当天，卡普尔过来找他，谈起关于 OSAF 的计划。当时卡普尔正犹豫是否采用开源方式来做他的新项目。过了几个月，赫兹菲尔德决定加入卡普尔的项目，做个全职志愿者。他不但是软件传奇人物，也是个音乐收藏家。他拥有几千张数字化唱片，还全情投入编制鲍勃·迪伦（Bob Dylan）[1]的歌曲编录工作中。换言之，他本人就是个活用例。

[1]美国民谣歌手。

赫兹菲尔德开始参加 OSAF 的每周讨论会，还很快着手做一个叫做 Vista 的程序，即卡普尔心目中那种灵活性极强的程序的原型。赫兹菲尔德会走开，写一些代码，再回到卡普尔的办公室，展示 Vista 新版本能干点什么好事。

Vista 是一个“面向用户”的程序：它注重向用户展示数据，注重输入、组织和修改数据的方式。所有信息都存储在 Shimmer 中——Shimmer 是摩根·萨奇基于 RDF 编写的数据库，或者按 OSAF 的说法，一个知识库。Vista 和 Shimmer 组合起来，让人可以管窥卡普尔梦想中的那种突破——一种让你可以按照自己设定而非计算机设定的规则管理生活中细节信息的工具。不过它还有局限性，而且经常崩溃。它只是个原型，用来尝试和总结经验教训，然后就会被丢弃。

对于任何采用新技术或新设计的项目，弗里德里克·布鲁克斯建议道，“准备丢弃一切吧”，因为你大抵不能一次就做对。你所能掌控的，不过是从一开始就有做错的**准备**，或者干脆“保证交付垃圾产品给客户”。



2002 年春，卡普尔更明确自己要做什么程序了。和 Agenda 一样，它也该是个 PIM（个人信息管理器），一种管理邮件、约会、地址簿、任务和备注的工具。但它应该能“跨平台”：不管你是用 Windows、Mac

或 Linux 都可以运行它——Windows、Mac 和 Linux 是现今最为普遍使用的三种个人计算机操作系统。而且它的架构设计要明晰，让任何开发者都能为其添加新功能。开源程序员可以编写新模块，或者说“包”，来增加功能，例如，管理数码相片或者音乐收藏。而非技术用户也能任意添加新类别或新标签，随加随有。

目标宏伟。卡普尔开始招聘开发者。第一位是斯坦福大学的毕业生，做事谨慎的程序员凯蒂·帕兰特（Katie Parlante）。在网络热年代，她曾在两家急起急落的创业公司呆过。帕兰特性格沉静平和、与人无争，但却有着敏锐的技术思维。只要她发话，别人都会倾听。在她安静的外表背后，是一颗坚忍不拔的心。随着时间流逝，她将稳步走向 OSAF 的管理核心。不过，现在她只是在快活地研究用户行为模式——例如，PIM 如何帮助人们绕过烦人的电话语音提示？又如，对于现有的日历软件，人们最集中的抱怨是什么？



凯蒂·帕兰特

帕兰特加入 OSAF 后不久，卡普尔聘请经验丰富的程序员约翰·安德森做 OSAF 的系统架构师。要定义“系统架构师”这一职位可不容易。有一回，在一次军方软件承包商的聚会上，当雷神公司（Raytheon）[①]负责工程的副总裁问及在场人士有多少认为自己是“系统架构师”时，一半人举起手来。还有三分之一的人认为自己是“系统工程师”。副总裁又问他们怎么定义“系统架构师”，怎么区分“系统架构师”与“系统工程师”。一时举座无语，有人作答后，其他人又表示不满。有些人被触怒了，他们似乎认为这个问题太唐突。最后，有个与会者站起来，背诵了一段 IEEE（the Institute of Electrical and Electronics Engineers，电气与电子工程师协会）标准文档中的文字，不过这段官样文章反而强调了副总裁的观点：满屋子依靠精细规范做事的专业人士们，没办法给自己的职位下个简单而明确的定义，也不同意别人的定义。

[①]美国军火制造商，产品包括爱国者、响尾蛇导弹系列，空中国王、比奇支线飞机系列等。

在 OSAF，约翰·安德森的“系统架构师”职位意味着他主要负责

选择其他程序员用来创建软件的部件：应该采用什么语言编写代码？应该采用什么工具来创建程序的图形界面——用户将看到的窗口、菜单和对话框？应该采用什么软件技术来存储程序数据？应该采用哪种数据交换标准？千头万绪有待抉择，而每个抉择的可选项也纷繁复杂。



约翰·安德森

安德森曾获动物学博士学位，整个职业生涯都在做软件，在麦金塔和 NeXT 平台上经验尤为丰富。在 NeXT 公司时，他曾负责管理工作，这段经历不乏挫折。“坦白讲，我管理得不好——我根本不清楚自己在做些什么，”他说道，“不过我学到很多东西。过去，我总以为最好的创意能赢，就因为它是最好的创意。不过现在我明白了，还有政治因素存在。那可是种全新体验，信不信由你。”在加入 OSAF 前，安德森也做过颇受尊敬的独立咨询师，为一些公司驻场解决棘手的技术问题。“就像是到一个不和睦的家庭做客——完了就走人，”他说，“你并不真属于一个组织，所以也不会纠缠到那些情绪化战争中。那时我解决了许多自己感兴趣的问题。”

安德森承接的项目源源不断，但在 2002 年初网络热降温、技术工业经济崩溃时，生意也陷入低迷。他打电话给朋友们，寻找合适的机会。安迪·赫兹菲尔德告诉他关于 OSAF 的事情。他回忆说：“那个项目和我手上自己的项目一样好玩。而且正处于最好玩的阶段——万事皆在计划，但什么都还没做起来。”

对于程序员来说，如同对于作家、艺术家以及所有那些在白板一块上开始描绘作品的人一样，项目“最有趣”的部分往往是在刚开始的时候。那时，有无数种可能敞开着大门，而通向痛苦妥协的大门却是关闭着的。

“程序员就像诗人，他的工作几乎全是纯思考，”弗里德里克·布鲁

克斯写道，“他运用想象，凭空造出城堡。唯有想象才如此灵活，如此易于琢磨和重塑，如此适于构思宏大的概念性架构。”

布鲁克斯这段话正与莎士比亚《仲夏夜之梦》中的著名诗句相呼应：

疯子、爱侣和诗人，
一总都是幻想造……
诗人的眼珠疯狂一转，
看尽天上人间，人间天上；
想象催生未知之物，
诗人的笔管
赋其形状，使空无者
也有了居所和名字。

这段台词是雅典公爵提修斯（Theseus）说的，他把“疯子、爱侣和诗人”都看作他们自己幻想的受害者。

提修斯推崇“冷静的理智”，鄙视“头脑发热”的“雕虫小技”。

不过我们可以这么理解他的台词：它给出了让梦想成真所必需的创造性工作的精准大纲。程序的确“几乎全是纯思考”的产物，但不会永远停留在思考阶段，否则就什么也做不出来了。程序员从思维的沃土上摘取点子，再用一行行具有实际功能的代码实现它——让它在计算机世界中“有了居所和名字”。



20 世纪五六十年代最早期的商业计算机，其存储器是铁氧体磁芯

（ferrite cores）。半导体芯片时代虽已不再采用电磁技术，但程序员们仍把活动内存叫做“磁芯（core）”。在许多系统中，当发生错误时，当系统因严重冲突或未预料的错误而死机时，就要做“磁芯转储（core dump）”——计算机，把出错那一瞬间的内存状态完整保存到一个文件中，让调试者有线索可循。

磁芯乃是一种物理实体，并非仅是比喻。它有居所和名字。你的计算机内存中有广大的空间容纳许多真实位置——在今天的高级芯片中它们处在微观量级——这些位置拥有名字和地址。在任意给定的瞬间，内存中的每个地址、每个比特，都保存着一个值为 0 或 1 的数据；这是种简单开关，要么开启要么关闭。计算机上运行的其他一切，不管是公司藏书还是毁灭者（Doom）游戏，都建立在这种二进制的基础之上——中央处理器（CPU）遵从程序员的指令，从内存中取出这些比特，进行一些操作，再把它们放回去。

软件则有不同：它没有磁芯。它就像洋葱般层层叠叠，每一层都辛辛苦苦地建立于前一层基础之上，危如累卵，指望着底下那层不要移动或改变太多。做软件的人喜欢讨论垒砖头；而怀疑论者眼中只看到空中楼阁。无论如何，日积月累，一层一层搭建起来。

程序员把这种结构叫做“抽象层叠”，每当新添一层时，就要把一些复杂而特殊的东西转换为简单而通用的东西。“抽象（abstraction）”一词来源于拉丁语，原意是“拉开”；在计算机科学中的定义是：“将多个较小操作组合为一个可通过其名称引用的单元的过程。”幼儿在托儿所就学会了抽象：他们会意识到，如果想吃苹果，其实无须手指苹果，只要讲出“苹果”这个词就行。他们也认识到，抽象依赖于一种通用语言，依赖于使用者共同假定“苹果”这个词就是指某类事物。

“这就是程序员做的事，”微软 Internet Explorer 浏览器开发组最早的负责人艾瑞克·辛克（Eric Sink）写道，“我们做了成堆的抽象体。我们

设计自己的抽象体，然后把它们堆到从别处拿来的抽象层叠上。”随着时间推移，抽象层叠越来越高。



在抽象层叠的最底端，正好在核心内存之上的部分就是汇编语言。汇编语言诞生于半个世纪前，是为了让程序员更方便地操作那些 0 和 1 而产生的。以前他们需要编写大量二进制代码，例如 1011000001100001，而现在则可以使用一些类似“mov”这样的命令编写具有同等效果的汇编语言指令，变量名也更加易于记忆。你还得手工操作比特，不过现在有了更方便的工具。

对于大多数人，汇编语言很难学习和编写。程序员兼评论家艾伦·乌曼（Ellen Ullman）对此曾有妙语云，汇编语言“离机器很近”——即便不在机器脑袋里，也是附在机器耳边说话。汇编语言普适性不强：每种 CPU 都有自己的不同汇编语言，要在不同的计算机上运行，就得重新编写代码。故此，昔时的程序员开始添加更多抽象层，每一层都离机器更远、离人类更近。

20 世纪 50 年代到 60 年代的十年间，可谓新编程语言大爆发的寒武纪。当时产生的许多编程语言，要么至今仍在使用，要么成了目前主流语言的先祖：例如 Lisp、Cobol、Algol、Basic 等等。（这些语言的名称都很技术化地缩写了，而且通常都用大写形式，我用大小写混拼，好让你的眼睛不至于太累。）第一个被广泛使用的是 Fortran——即“FORmula TRANslating system（公式翻译系统）”。Fortran 把编写汇编语言的费力过程概括为简短的命令。程序员编写一系列命令——即源代码——然后计算机上运行的一种叫做“编译器”的用户程序把这些命令翻译为机器语言中的“对象代码”。就像是只会挨个读出字母来沟通的人突然之间学

会了连续发音一样。

这是一种飞跃，这种飞跃甚至让 Fortran 的鼓吹者们相信，计算机编程从未取得过如此彻底的超越。关于这项发明，他们写道，将“在本质上消除编码和调试。”他们吹嘘 Fortran 是“自动编码系统”，因为它取代了机器语言中晦涩的二进制代码或稍具可读性的汇编语言，代之以可读性相对较高的命令序列，如 READ、ASSIGN、GOTO 和 STOP。最初，Fortran 看似展示了一幅美景，使得让计算机屈从人类的艰苦斗争——如 Fortran 的发明者约翰·柏柯斯（John Backus）形容原始编码所言，“一场与机器的肉搏战”——化为乌有，让人类可以通过这种魔法般的系统，用自己的语言直接向机器诉其所愿。

然而，如同随后几十年里编程领域的其他发明一样，梦想并未成真。归根到底，Fortran 的全部创见和功用，在解决了一系列问题的同时，又带来了其他问题。在 Fortran 问世后的十年中，整个编程领域都成为其成功光环背后的受害者，专家们甚至召开了一次国际性的秘密会议，商讨如何能逃离“这场软件危机”。



Fortran 大获成功的原因之一是，它挑战了专家们关于这类“高级语言”运行起来一定比手工编写的机器代码慢的预言。当你从层叠的底部往上移，离机器越远，你就是在要求计算机干越多工作；在使用 Fortran 时，在计算机运行你的程序之前，首先要让编译器执行一遍源代码。不过，只要这么做一次，编译器生成的代码就会变得相当高效：Fortran 编译器“自动”生成的代码，被证明与程序员用机器语言或汇编语言编写的代码一样好。对于计算机科学家们而言，这意味着 Fortran 编译器效率惊人、接近完美。只要多花一点点额外的计算机时间来执行 Fortran 程序，

就能节省程序员的大量时间。

在 Fortran 发明的时代，计算机时间相当昂贵：机器稀少、体积庞大、贵得离谱。不过变化已在发生。大约与 Fortran 发明同时[①]，1965 年，Intel 公司创始人戈登·摩尔（Gordon Moore）提出了他关于计算机硬件功率高速发展的著名预言：摩尔发现，在单块计算机芯片上集成的晶体管数量，在每一、两年内，以指数速度增长。尽管摩尔调整过几次具体数据，不过基本观点却保持一致。持续翻倍的数字会迅速变大。单块芯片上的晶体管数量翻倍，意味着你能以双倍速度运行同一计算机指令，或者说运行成本降低一半。拜此种现象所赐，在摩尔的观察结论提出之后的四十年里，计算机硬件每年都越来越快、越来越便宜。

[①]原文如此，实际上 Fortran 是在 1957 年发明的。

对于程序员们而言，这意味着计算科学年轻时代的积习——那时你得微调，或者说“优化”代码，尽可能降低内存使用量和“周期”（处理器时间单位）——在未来将不复必需。过了许多年，当卡普尔和 OSAF 开始他们的工作时，处理器时间明显已经比程序员的时间便宜得多。许多认真的软件开发者——如同惜字如金的作家，或者物尽其用的厨子——仍然秉承节约和自律的美德。然而，当让计算机还是程序员更努力工作的选择降临时，还是令人难以决定。

那是因为软件领域没有摩尔定律。芯片性能可以每一、两年翻一番；但是大脑的能力可无法如此倍增。



当约翰·安德森、安迪·赫兹菲尔德和米奇·卡普尔在 2002 年夏天选择 OSAF 要用的编程语言时，他们都很清楚这段历史：过去 30 多年的个人计算机业界，他们有所经历。自从 Fortran 语言发明以来，众多编

程语言构成的巴别塔（Tower of Babel）[①]确已矗立，这些语言既坚固又脆弱。有些较适于编写商业应用软件，有些则适于编写科学应用软件；有些适合短平快的项目，有些适合实现庞大构想。每种成功的语言都有一群死忠分子。他们之间的争论往往类似于教派之争，甚至在重大问题上也不可化解。编程语言设计的核心冲突总是有如弥尔顿式的两难抉择[②]，一边是自由意志，一边是圈定范围。例如：是否应该赋予程序员直接操作计算机内存的能力——同时也带来使计算机崩溃的危险？或者应该划出安全区，减少错误发生的可能性——同时绑缚了程序员的双手？

赫兹菲尔德使用 Python 语言来编写 Vista 原型。Python 发明于 20 世纪 80 年代后期，发明者是德国程序员圭多·范·罗萨姆（Guido van Rossum）。Python 得名于英国荒诞喜剧团巨蟒的飞行马戏团（Monty Python's Flying Circus）。（巨蟒剧集的无厘头解构风格在计算机实验室中总有拥趸；我们把不请自来的垃圾邮件叫做“spam[午餐肉罐头]”，是因为互联网先驱们在给这种玩意儿命名的时候，想到巨蟒剧集中演过一个小餐馆，他们的菜单上只有“鸡蛋、香肠、午餐肉、午餐肉、午餐肉和午餐肉。”）

Python 是一种“解释型语言（interpreted language）”。“编译型语言（compiled language）”通过编译器先将程序员的源代码翻译为机器可读的二进制代码后再运行，而解释型语言则是在运行时做类似的工作——解释器逐行翻译源代码，再喂给处理器运行。解释型语言效率较差，因为你要同时运行自己的程序和解释器。但这也使得解释型语言较为敏捷。程序员不用等到编译器翻译完所有代码即可看到运行效果，而程序也能够在运行时响应修改。编译型语言和解释型语言的区别，就像是你要到斯洛波维亚[③]旅行，是带一本双语对照指南书还是一位翻译人员的区别那样。雇翻译增加了花销，不过如果不幸遭遇地震或政变，有个斯洛波维亚专家在身边总会好些。

[①]《圣经》创世纪 11 章载：人类要在示拿地造通天塔，扬自己的名。耶和华变乱他们的口音，使语言不通，造塔之事遂不成，故称巴别（即“变乱”之意）。

[②]弥尔顿在《失乐园》开篇即写道，撒旦战败后被囚于地狱，仍然坚守其“无法击败的意志”，“宁在地狱为王，胜过服事于天”。

[③]Slobovia，美国少年 Troy Thompson 于 1994 在明尼苏达州创建的虚拟微国家（micronation），拥有自己的语言（类似德语、英语混合体）。现在常用来代指某个远方国家。

Python 和其他解释型语言一样，为许多软件开发者所轻视，他们把解释型语言看作“脚本语言（scripting language）”——只用来临时拼造短程序脚本，编点快速工具代码，不适用于较为严肃的项目。但到了 2002 年，Python 的拥护者们开始挑战这种假设。

Python 新版本的产品经理是刻薄的澳大利亚程序员安东尼·巴克斯特（Anthony Baxter）。他在介绍一个基于 Python 的互联网语音通话项目时，幻灯片上写着：“‘脚本语言’，我那亮闪闪的铁屁股！”（他是在向马特·格洛宁[Matt Groening]《未来狂想曲[Futurama]》[①]中那个坏脾气机器人笨德[Bender]的名言致敬。）安迪·赫兹菲尔德也为 Python 大唱赞歌：他说，自己在用 Python 时生产力要 3 倍于用 Java，而用 Java 的效率又 3 倍于用 C。C 是个人计算机业界和大学里面大多数人都要硬啃的语言。Python 解释器是用 C 写的；这使它成了新一层的“高级语言”，上了一个台阶，离机器更远，离人更近。

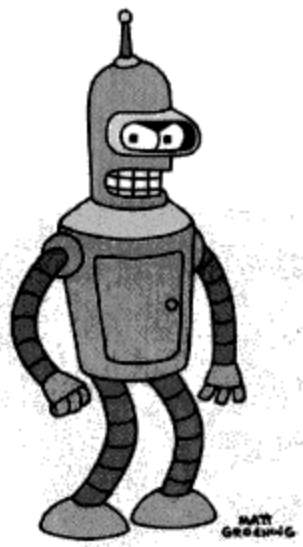


图 3-1 机器人笨德

使用类似 Python——或其老对手 Perl——等高级语言编写的代码，往往比用较底层的语言编写的代码更为紧凑简洁；许多东西被抽象掉了。范·罗萨姆说，完成同样的任务，Python 代码要比 C 或 C++代码少上 3

[①]格洛宁是热播 15 年的动画剧集《辛普森一家（The Simpsons）》的缔造者。1999 年，Fox 电视台邀请他制作了科幻动画剧集《未来狂想曲（Futurama）》。该剧集处处调侃经典美国科幻，充斥犯罪、暴力、性及粗口。

倍、5 倍、甚至 10 倍。这不仅代表程序员可以少敲许多次键盘；还意味着当程序员回头添加新特性或修正缺陷时，更容易理解某段代码的功用。“我确信，每个程序员都能在大脑中记住一些代码，或者把这些代码放在文件中，便于管理，”范·罗萨姆说，“这些代码大致体现为一行行语句，而不是代码真正干的复杂工作。所以，如果有 5 倍的代码，就的确需要花 5 倍的力气去维护它。”

撇开多次给 Python 语言改名不谈，范·罗萨姆基本上是一位严肃的实用主义者。某次他用一张幻灯片向一群程序员介绍自己的生平时，巨蟒迷的神髓悄然显现：

4 岁：第一套乐高积木

10 岁：第一套电动玩具

18 岁：第一个计算机程序（用打孔纸带做的）

21 岁：第一个女朋友 :-)

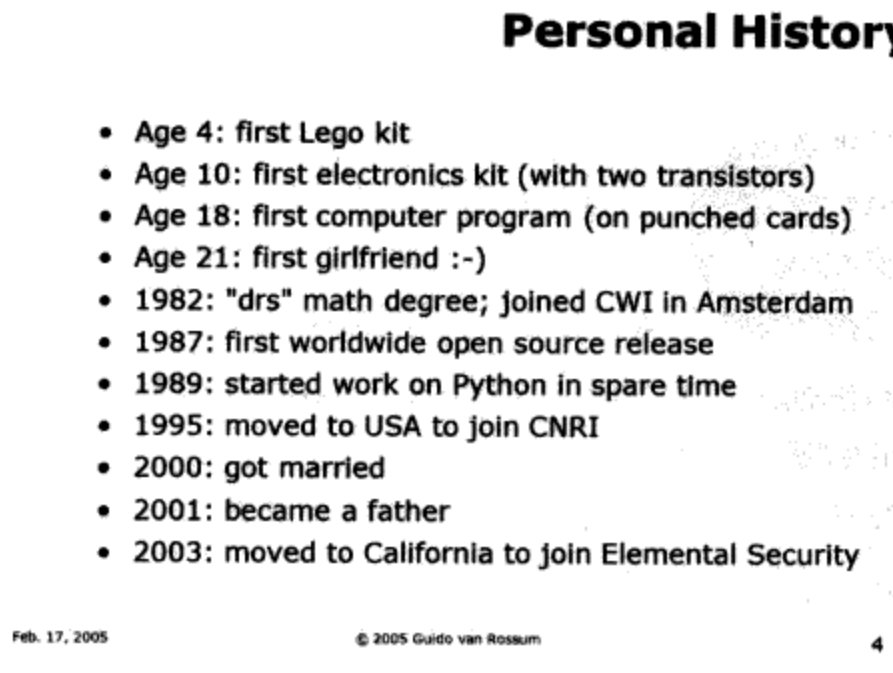


图 3-2　圭多的幻灯片

就像宠物狗会长得像主人，看来编程语言也微妙地反应了其创造者的气质和性格。比如，Perl 之父、语言学家拉瑞·沃（Larry Wall），就热衷于口头和书面双关语。在 2004 年一次开源程序员的聚会上，拉瑞·沃回顾了一年来 Perl 新版本的开发，他将屏幕保护程序呈现的几何变幻图形与自己治疗胃部肿瘤的账单扯到一起，然后还扯到 Perl 程序员社群"随机相互展示美"的愿景。

所以，Perl 语言如此之重视表达式的灵活性，也就不足为奇了；Perl 的信条之一是"There's more than one way to do it（大路绝非只一条）"。（喜欢缩略语的程序员们给这句话加了个抽象层，把它变成了 TMTOWTDI。）不喜欢 Perl 的人认为，可以有多种方式达到同一结果的自由，也让程序员们不易读懂其他人的代码。而 Python 的信条，范·罗萨姆说，是 TOOWTDI：There's only way to do it.（仅此一途。）有好事者总结这两种语言程序员的特点，说 Perl 程序员是"奇技淫巧之徒"，而"Python 程序员则崇尚和平，就像那种用条细线系眼镜的霍比特人[①]或者荷兰人。"

[①]托尔金《魔戒》系列小说中的小矮人族。

Perl 和 Python 的拥趸之间多年以来争斗甚烈，不过无论大争小斗，结果多半是好玩多过痛苦。范·罗萨姆和沃于 20 世纪 90 年代中期相识。"Perl 和 Python 社群当时已经开始有冲突，不过拉瑞和我相处得还算融洽，"范·罗萨姆说，"我们都不关心也不担心如果其中一种语言'打赢'会怎样。我们都不太在乎输赢。我们不做那种竞争。"

像 Perl 和 Python 这样的语言在 20 世纪 90 年代互联网搞增长时期颇为盛行，紧张的项目时限迫使各公司采用这类"脚本语言"编写"胶水代码"或"胶带程序"，把 Web 服务器和数据库绑到一起，构成可运转的网站。不过它们仍被看做是轻量级的。例如，有些程序员就见不得 Python 中空白符的使用方式：很多语言都用括号或其他符号来划分代码块，而 Python 却只简单地用缩进表示。这让 Python 代码较易阅读，但

也较为脆弱，因为缩进在电子邮件中会消失，当你将代码转移到其他操作系统时，也可能发生改变。

Python 另一引起争议的特性是其对变量“类型”的规定，这是相对较深奥的技术问题。编程中的变量类似代数课黑板上写满的“x”和“y”，但是也不尽同。变量是数据的代指，是计算机内存中空槽的名称，这些空槽会由用户输入（或程序另一部分输出）的信息所填充。例如，在银行程序中，当你输入开户金额时，程序就会把这个数字赋予一个类似名为 *balance.starting* 的变量。程序可以随机对变量赋值或要求用户输入。然后把值存储到计算机内存中。下次如果程序代码提到 *balance.starting* 时，计算机就会取出那个值。

有些编程语言在变量类型方面设计得较为严格，当程序员创建变量时，会被要求同时声明变量的类型：是整数？是十进制浮点数？还是包括各种字符的字符串？有些语言则较为宽松。有些语言在编译时强迫遵守这些规则；有些语言直到运行时才遵守。术语“*静态类型（static typing）*”、“*动态类型（dynamic typing）*”、“*强类型（strong typing）*”、“*弱类型（weak typing）*”正是用来描述不同的类型规则，不过这些术语常被误用，也被人们错误地理解——它们自身就是弱类型的。有位名叫本杰明·皮尔斯（Benjamin Pierce）的计算机科学家撰写了一系列关于这个话题的著作，他写道，“我花了几个星期想要理顺诸如强类型、静态类型、安全类型等术语，到头来发现奇难无比。它们的用法如此多变，实在无从着手。”

尽管存在这种混淆，或者正因了这种混淆，在程序员邮件列表和论坛中变量类型的话题常常引来板砖横飞。变量类型看似小事，但在代码的国度却能引发宗教战争般的冲突。批评者鄙视 Python 在变量类型上的宽松规则；但其拥护者则认为这反而带来了比其他语言更多的智能特性。不管怎么说，Python 的解释型特性使其很容易被看做是较不严肃的语言。

不过，在另外几个主要方面，Python 却比其他一些前辈脚本语言更符合业界标准的编程语言。首先，它提供了“垃圾回收（garbage collection）”机制：当程序不再需要先前申请的内存空间，Python 就会负责清理，就像在 Java 以及其他同时代语言中那样。（很多较早的语言要求你手工管理内存，把程序员当作了旅馆前台服务员，应付内存中的临时信息房客。）还有，跟 Java 和 C++一样，Python 也“面向对象（object oriented）”。

且说“面向对象”：和许多其他远离程序员封闭世界的密语不同，这个术语模糊不清，常常导致理解错误。即便是读过数百页资料、和专家几番探讨、再听上一堆冗长的讲座，我还是不敢百分百相信自己关于这个词组的解释能让所有读者满意。来吧：面向对象技术不是围绕一系列命令行来组织程序，而是基于一些叫做对象的代码片段来组织程序。当接收到其他对象的呼叫时，对象就会做出操作。对象之间通过严格定义的输入输出彼此相关，所以，程序员可以编写与对象交互的代码，但不必了解对象内部的运作机制。面向对象编程给软件领域带来了一系列新词汇：软件对象通过互相之间发“消息（message）”来触发“事件（event）”，达到交互的目的。通常，共享特性对象可以组建为“类（class）”，还可以从“父类（parent）”继承特性。（这种命名法有时让关于面向对象编程的讨论像是马克思理论和遗传科学的混杂。）

面向对象编程的历史可以回溯到 20 世纪 60 年代，但直到 20 世纪 80 年代才横扫市场。那时，程序员开始从命令行界面的串行世界——一种文本行环境，用户得逐行输入和计算机的交互命令——迁移到图形界面的混乱天地，用户可能在任何时候点击任何位置（“触发一个事件”），而程序就像一位聚精会神的守门员，需要随时做好扑球的准备。面向对象编程的原则看来像是管理这个混乱世界的理想工具。

在面向对象编程流行起来之前，技术媒体就已经关注对象的概念，并过于简单地将它解释为“可复用的代码段”。软件对象——将如同

Fortran 和其他许多先前的创新那样——改变一切，它将把编程从宏图大业变作从类库中“借用”组件的零敲碎打式的工作。可是，尽管面向对象方法能在程序员构筑更为复杂的大型系统时助他们一臂之力，但它仍未开辟出通向销售人员描述的乌托邦的道路：在那理想之国，程序员只要写一次组件，就能到处复用。代码越来越多——到处是对象库、函数库、模块库，许多代码遵循开放源代码授权，让你可以随意采用、自由组合。在打造大型新项目时，程序员可以站在也许更高的巨人的肩上。但他们仍然要自行决定何去何从。

就像弗里德里克·布鲁克斯在《没有银弹（No Silver Bullet）》中所说的那样：“构建软件系统最难的就是精确设定要做什么东西。”



卡普尔及其同事有很多理由采用 Python。Python 开源而且跨平台（同一个 Python 程序可以在 Windows、Macintosh 和 Linux 操作系统上运行），就像他们要做的那个程序一样。有些新语言也有其吸引力所在——例如由日本程序员发明的 Ruby 语言，就吸引了不少眼球——但 Python 更为成熟。不仅因为它历经磨炼、剔除了许多缺陷、改善了运行效率，还因为程序员们为它制作了大堆工具——新的抽象层和辅助程序让 OSAF 的程序员无须从头开始。如 Python 的拥趸们所言，这种语言“内装电池”。Python 是用机器时间换取开发者时间，所以它肯定不如其他一些语言快，但它也允许你在代码中找到瓶颈并优化之——你可以往下“接近机器”，用 C 语言重写效率欠佳的部分，效率与方便兼得。Python 迷们满怀敬畏之心谈及“圭多的时间机器”：每当程序员们抱怨 Python 的某个缺点时，范·罗萨姆总是指出，新近的版本已经修正了问题——于是，关于他能回到过去对他的造物进行及时修改的笑话就流传开了。

在技术圈总有安全的选择——像那句老话讲的，“买台 IBM，不会被解雇。”对于 OSAF，“不会被解雇”的编程语言选择就是 Java。Java 能够满足卡普尔团队的多种需求——跨平台、面向对象；商业人士接受它，程序员也很了解它。但它并非真正开放源代码（太阳微系统公司拥有其代码），而且卡普尔用 Java 编写 Agenda 新版本的经验也让他质疑 Java 是否可以写出那种他梦想中漂亮而直观的图形用户界面。

在这方面 Python 也并不完美。约翰·安德森和凯蒂·帕兰特检视了一系列被称作 GUI 构造器的跨平台 Python 工具集——用于构造窗口、菜单、按钮和其他屏幕上的“部件”，让 Python 程序可以在 Windows、麦金塔和 Linux 用户面前展示图形界面的要素集。其中一个叫做 wxWidgets/wxPython 的构造器脱颖而出（最早叫做 wxWindows，后来项目组把 Windows 字样去掉了，以免和微软发生法律纠纷）。Wx 当时尚未完备——麦金塔版本尚在开发中——不过安德森认为再加把劲就能让它为 OSAF 工作。

开发者们也短暂考虑过采用 Mozilla 项目组开发的工具集。Mozilla 是在网景废墟上建起的开源网页浏览器。Mozilla 拥有能在 3 种操作系统平台上正常运行的 GUI 构建技术，而且有好几年的开发历史。追本溯源，它最适于用来编写在网页浏览器中运行的程序。这类程序一直被看做是鸡零狗碎的玩意儿，直至 2002 年，软件业界领导者们开始意识到它的妙处。它能帮你解决跨平台问题（浏览器本身就有针对不同流行操作系统的版本），而且用户可以把数据保存在服务器，随需拉取到要运行的计算机，不管计算机是在家里、在办公室、还是在车上。

卡普尔对这些情况了如指掌，还做了仔细的推敲，不过最终他还是认为在浏览器中不可能实现他想要的那种丰富的交互操作，以及从“Agenda 之魂”继承而来的灵活性。至少当时不行。限制太多了——在屏幕上拖放及操作控件、在窗口中快速更新数据等等，都有问题。

基于网页的程序太过沉闷，而卡普尔永远不想让人觉得自己的作品“沉闷”。



选项太多，往往导致软件项目在选择编程语言时随性而为——根据个人品味、习惯或者就是一时心血来潮。程序员评论家保罗·格雷汉姆（Paul Graham）写道，有些程序员选用 Python，只是因为他们喜欢 Python 代码的样子——而且也不无道理：“编程时，读代码的时间要远远多于写代码的时间。你对待源代码的方式，和雕塑家对待黏土的方式一样。所以，一种语言如果让源代码显得丑陋，就会叫苛刻的程序员发疯，仿佛雕塑家遇上了满是硬块的黏土一般。”

尽管这需要极大的信心——怎么也不能用“脚本语言”来做一个目标宏伟的桌面软件的基础吧！——但卡普尔、安德森和他们的团队最终还是选择了 Python。感觉不错。押宝在速度上——这种语言本身能给予 OSAF 程序员的额外速度，以及摩尔定律给予最终用户机器的额外速度，希望能抵消 Python 自身的性能疲软。

OSAF 的每个人都很重视速度，因为 2002 年的工作进度正在停滞中。当更重要的架构级讨论开始时，赫兹菲尔德不再往 Vista 原型中添加新特性，转而描绘新产品的地址簿或“联系人管理器”的表现形式。帕兰特停止了用户调研，开始编写日历原型。摩根·萨奇着手为项目创建一套叫做 Hardhat 的自动构建系统——一种代码框架，帮助开发者在修改代码后重新组合程序的构建块。每个人都急于开始编写代码。

但直至秋天，OSAF 仍然悄无声息。卡普尔是个公众人物；硅谷传言他就要做点什么了，但到底要做什么则无人知晓。卡普尔不肯在有可工作的产品之前就大肆宣扬——那就成了典型的“雾件”，重蹈 20 世纪

80 年代后期他创办的 On Technology 公司的覆辙，当时《麦金塔（Macintosh）》杂志专栏作者将 On Technology 的产品列入了“雾件名人堂”。可是，假使你打算做一个开源软件，就不可能秘密进行；项目要公之于众，这样才能吸引有兴趣的志愿者加入。卡普尔的朋友，《圣何塞水星报（San Jose Mercury News）》著名技术专栏作者丹·基尔默（Dan Gillmor）将他推向公众。一开始，卡普尔还百般抗拒，随后他意识到这或许是件好事——一种让 OSAF 开动引擎的“推动力”，逼着他们去解决项目中一些始终悬而未决的不确定因素。比如项目的名称。

软件业长久以来习惯于为开发中的产品取代号。在商业世界中，通常先用程序员给出的这类替代名称，稍后市场人员再想出一个最终的“发行”名称。而在开源世界中，程序员说了算——所以许多开源产品的名称都用了拗口的技术性双关语、隐晦的典故或者冷笑话。（采用最广泛的 Web 服务器 Apache，其名称就来自于它“逐个打补丁[patch]”的面世过程。还有类似 Mozilla、Knoppix、Blojsom 等名称。）一般来说，项目的利润越高，你就会越谨慎地选择项目代号——微软在 2001 年曾有过教训，当时他们将一套个人信息管理系统命名为 Hailstorm（冰雹），结果招致猛烈抨击。

卡普尔不想浪费大量时间给 OSAF 项目取代号，他尽量推迟做这事。那时，摩根·萨奇正在为 OSAF 搭建一个在线平台——一种叫做 CVS（Concurrent Versions System，并发版本系统）的代码库，好让开发者在工作时能“签出”代码、完成后再“签入”对代码做出的修改，并且能跟踪这些事件。CVS 这样的版本跟踪系统让分布在不同工作地的程序员能基于同一代码库同步开工，而且不会发生版本冲突；版本跟踪系统让开放源代码开发有可能实现。9 月的一天，萨奇找到卡普尔说，为了建立 OSAF 的 CVS 代码库，他需要给新项目的文件目录取个名字。卡普尔想了几秒钟，说：“就叫 Chandler 吧。”

“我爱看钱德勒（Chandler）[①]的推理小说，百读不厌，”后来他这样解释道。用一位推理小说家的名字来给项目命名，意味着其他小说家的名字——哈迈特（Hammett）[②]？克里斯蒂（Christie）[③]？——也可能会被用来给以后 OSAF 的其他项目命名。这文绉绉的名字，与通常借用技术圈行话的做法格格不入。而且，用一位经典推理小说家的名字来命名，也带有一种对该项目的保密状态表示赞同的幽默意味。

萨奇有点担心人们不会想到那位小说家，而会联想到电视系列剧《老友记（Friends）》[④]中也叫钱德勒的那个角色。不过他没提出反对意见。Chandler 就 Chandler 吧。



2002 年 10 月 20 日，基尔默在《水星报》的专栏上向外间介绍 Chandler。同时 OSAF 也开放了一个网站，卡普尔开始写个人 blog，记录项目大事记。如卡普尔所愿，项目公之于众果然迫使 OSAF 更清晰和确切地说明 Chandler 到底将何去何从，把模糊不清的想法提炼成硅谷人称“电梯游说（elevator pitch）”的简短说明——所谓“电梯游说”，就是当你有幸在电梯间遇到某位权钱人士时，能脱口而出，在短时间内说服他。Chandler 将会是一个跨平台、开源、P2P 的个人信息管理器（PIM），重点功能是电子邮件和日历——全面贯穿来自 Lotus Agenda 的动态主义精神。这话解释起来得费一番工夫；看来那电梯最好再多上几层。不过这足以让关注此事的软件狂人和开源分子兴奋一时了。

关于米奇·卡普尔正在致力于一种全新的“人际信息管理器”的消息震惊了奇客圈。基尔默的消息散播甚广——网上最具人气的程序员窝点 Slashdot 也发布了链接。数百位有心人在 OSAF 刚开放的邮件列表上贴出了自己的想法和期望。多数邮件可以归结如下：

- 我现在就想用 Chandler——哪儿能搞到？不会还没写出来吧？

[①] 瑞蒙德·钱德勒（Raymond Chandler），美国推理小说家，代表作有《菲利普·马洛》系列

[②] 达希尔·哈迈特（Dashiell Hammett），美国推理小说家，代表作有《玻璃钥匙》等

[③] 阿加莎·克里斯蒂（Agatha Christie），英国推理小说家，代表作有《神探波洛》系列

[④] 又译《六人行》

- 为什么不用这个或那个开源项目作为 Chandler 中这个或那个元素的基础构件呢?
- 我只会考虑用和我多年以来习惯使用的 PIM 具有完全相同特性的软件，否则免谈。
- Outlook 闪开。微软滚蛋。

Slashdot 的头条标题“米奇·卡普尔的 Outlook 杀手”定下了舆论的基调：卡普尔将取代比尔·盖茨。Chandler 将干掉 Outlook。要么和开源一起上天堂，要么和微软一起下地狱!

引起这样的反应在预料之中，但却让卡普尔备受煎熬。他明白，攻击微软就是在给自己找麻烦——微软产品已深入人心，而且这家公司也惯于在感到受威胁时出手把竞争对手打个稀巴烂。他认为自己很清楚地表示过，初始市场并非 Outlook Exchange 占据的大公司，而是更为“信息密集”的个人用户和小公司。他也曾说过，Chandler 没有取代 Outlook 巨兽的打算，只是想与之共存而已。不过，他低估了自己神话般的个人经历对项目的影响。一旦有卡普尔对决盖茨的些微迹象，媒体就会紧追不舍。

卡普尔知道，Chandler 要取得成功，不能靠攻击微软巨龙，而要靠用自己的优点赢得用户。长路漫漫，虽然代码尚少，但吉兆已示。有一些十分聪明和经验丰富的头脑正在着手解决他们为之着迷的难题。他们站在其他开源项目的肩头。谁可阻挡?

基尔默在 10 月 20 日的专栏中写道：“年底会在网上发布该软件日历功能的早期版本，整个软件的 1.0 版预计于 2003 年底或 2004 年初推出。”卡普尔在自己的 blog 上写道：“今年底我们将先向开发者开放代码以供查看。只是部分代码的早期版本。乐观估计，2003 年底能完成 1.0 版。悲观估计，可能要到 2004 年。”不过，他提醒读者，这是 OSAF 的“原始标准响应”，他自己对此“并不完全同意”，而且“鉴于 Chandler 目前的状态，能进展多快还不得而知，也不知道距首次发布还有多远的路要走。”

他的预估太过乐观了。

第 4 章　乐高王国

[2002 年 11 月～2003 年 8 月]

2002 年秋天，技术工业的过山车停在了令人沮丧的轨道最低点。网络股泡沫刚刚消失。硅谷许多沿街办公园区的停车场都变成了无人地带。后 9/11 时代的经济停滞了一年之久，旧金山街头贴着的“美国欢迎商务合作”的海报像是在自打耳光：说得倒好，可谁会买账呢？

技术工业的冰河时代导致大量程序员失业或半待业。相当多的程序员转而投身开源项目，以期维持自身的技能和在外界的露面率，同时挠挠自己的技术之痒。大批程序员赋闲，孕育了许多新软件——有小型项目，也有目标远大的项目，在业界的轮子重新开始转动时，这些大项目将成长为新一代的创业公司。

在这个特殊时期 OSAF 当然获益匪浅：它就像是照在黑暗原野上的一盏明灯，召唤着寻求项目的程序员们。米奇·卡普尔总认为，OSAF 既要依靠全职雇员，也要依靠无私的志愿者，而安迪·赫兹菲尔德就是后者的典范。基尔默专栏文章引来的新闻飓风给卡普尔刮来了大量

鼓励和愿意提供帮助的信息。史蒂夫·乔布斯打电话给卡普尔——这是10年来他们首次通话——邀请OSAF团队到苹果公司面谈，看看能怎么合作。


罗·蒙特利

当罗·蒙特利的名字在OSAF刚开放的“dev”邮件列表上出现时，引起了卡普尔的特别注意。罗·蒙特利建议OSAF再考察一下是否采用Mozilla工具集构建用户界面。蒙特利及其哥们阿历克斯·托蒂克（Aleks Totic）看过Slashdot上关于OSAF的文章；很快他们就联系上卡普尔，和他商量志愿为Chandler工作的事情。这两位年轻人在网络圈都很有名：托蒂克曾经在伊力诺伊大学香槟分校的一个团队中工作，该团队创建了第一个流行的图形网页浏览器Mosaic（托蒂克负责编写麦金塔版本）——之后又在1994年和一帮同事杀到硅谷，成为网景公司的创业元老之一。蒙特利在堪萨斯大学读书时就编写了基于文本的早期网页浏览器，后来应募到网景公司参加创业。他最出名的事迹应该是发明了“cookie”——一种网站通过浏览器植入用户计算机的文件，用来在会话与会话之间跟踪用户及其数据。cookie为人所非议，也常被误解。1995年8月，网景公司股票甫一上市就火箭般飞涨，开启了互联网空前繁荣的时代。和其他网景创始人一样，蒙特利和托蒂克也陡然暴富。好友喜共事，他们在网景公司快关门大吉时挂靴而去，和别人一起创办了野心勃勃的电子商务网站Epinions，当公司陷入低迷的漩涡中时，他们就都离开了。

2002年，他们发现自己无所事事到了极点。当时罗多数时间呆在太浩湖（Lake Tahoe）[①]，滑雪自娱。阿历克斯也大抵如此。Chandler听起来很酷。或许卡普尔需要帮助呢？

[①]加州和内华达州之间的高山湖，海拔6000英尺以上，积雪期长达8个月。

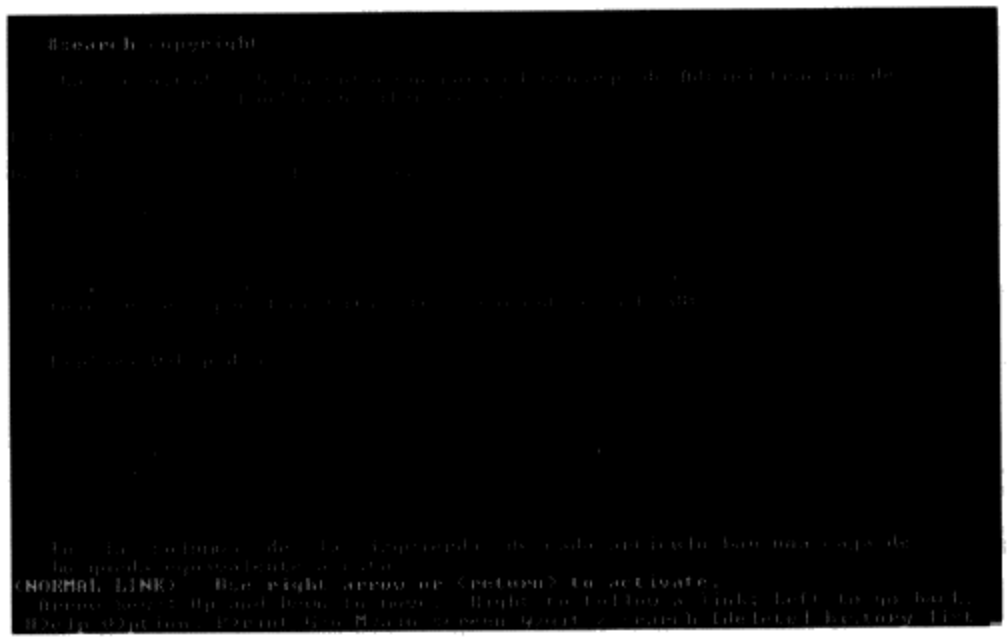


图 4-1　罗·蒙特利打造了早期 Lynx 和“cookie”

卡普尔知道，蒙特利和赫兹菲尔德及约翰·安德森一样，都是“搞应用的老伙计”——他以前写的程序，都是用户在自己计算机上单机运行的。但 Chandler 则与此不同：通过网络共享数据是关键。蒙特利和托蒂克曾经用了将近 10 年时间编写互联网的软件，在这方面他们自有一套。卡普尔想，这倒用得上。

托蒂克和蒙特利作为志愿者加入了 OSAF，同时卡普尔也开始为业已曝光的 OSAF 招聘其他职位人员。这支队伍中很多人曾在苹果公司做过，或许他们在卡普尔对未来的展望中看到了史蒂夫·乔布斯“改变这世界”的热情。林智超（Chi-Chao Lam）当上了 Chandler 项目的产品经理。他是个经验丰富但又童心盎然的硅谷企业家。林曾为凯兰达实验室（Kaleida Labs）[①]工作，之后加入 IBM 和苹果在 20 世纪 90 年代中期组建的命运多舛的合资公司，然后足具远见地创办了一个基于 Web 的广告网络。资深麦金塔电脑市场记者、行业分析师皮耶特·哈特索科（Pieter Hartsook）执掌 OSAF 的市场及公共关系部门。还有两位程序员加入编码组：刚从斯坦福大学计算机科学系毕业的学生杰德·伯吉斯，他和约

[①] IBM 和惠普合资成立的公司。

翰·安德森一起做 GUI 方面的工作。曾为苹果和网景工作过的程序员戴维·麦卡斯科（David McCusker）负责 Chandler 数据存储部分的开发。

从软件抽象层的另一种维度来看，软件开发者常常谈及“前台”和“后台”。用户在前台，前台是程序中和用户打交道的部分——显示窗体、对话框和鼠标指示，告诉你正在发生什么，让你有办法输入信息并得到输出信息。后台是前台发生的事件和用户输入流向的地方，计算机对事件和输入进行处理、保存、取回。（在 Chandler 的原型系统中，Vista 是前台，Shimmer 则是后台。）前台应该精致、直观、功能强大；后台应该隐形、高效、如岩石般强固。前台与人对话；后台与比特对话。用《星球大战（Star Wars）》中的人物做类比，前台就是细致周到的深思皮奥（C3PO），后台则是不可索解的糊涂迪图（R2D2）。

对于 Chandler 的前台，卡普尔比较清楚自己想要什么，以及它如何体现 Agenda 之魂。不管 Chandler 的后台做成什么样子，都要满足卡普尔关于前台的想法。但到底该做成什么样子呢？当 OSAF 的程序员们——例如凯蒂·帕兰特，她的日历部分代码是 Chandler 项目最早实现的功能代码段——努力开始编写程序，他们总会撞上没解决的问题：Chandler 的资料库——程序的数据存储仓库——是否应该采用 RDF“三元”格式，就像萨奇在 Shimmer 原型系统中那样？如何保存用来构造 Chandler 的 Python 对象——也在资料库中放置可行吗？如果 Chandler 要支持 P2P 数据共享，资料库如何管理这些数据？如果 Chandler 资料库保存在家里的计算机中，而你又想要在办公室用到这些数据，该怎么办？

怎么办？程序员们明白，如果没有得到回答，就无法进行下去。他们停步不前——用软件业的惯用语来说，他们被锁定了。

Chandler 的后台工作陷入了做出艰难技术决定的迷宫，而麦卡斯科猛然间发现自己正身处迷宫中央。麦卡斯科曾经在自己的“树龙（Treedragon）”blog 上记载开发新编程语言的项目，他加入 OSAF 后，

又开始在 blog 上满怀热情地逐封回复探讨技术的邮件。有封匿名邮件这样写道：

请别重复发明 Python 和 Zope（一个基于 Python 的项目）开发者已经创造出来的东西。Python 和 Zope 经过多年开发，积累了大批绝佳的技术财富。做好 Chandler 的关键在于复用，而不是重复发明。把前人的成功经验集成进来，不要重复发明。看看用多么少的代码就可以写出 Chandler 吧……

麦卡斯科回复道："我们打算尽可能多地复用现有代码，少写新代码。要加快工作进程，就得尽量避免踏入新的编码地带。"

然而，泛泛而谈代码复用是一回事，具体去选择采用已有代码片段是另一回事。每个代码块都优劣参半；在参加者逐渐增多的 OSAF 会议上，对每种实现手段也都是毁誉参半。谁也不想做出错误决定。从 11 月到 12 月，会议一个接一个像马拉松般召开，一直到过完新年，Chandler 团队艰难地向着达成一致的目标前进。

摩根·萨奇编写基于 RDF 的 Shimmer 资料库时，是当作原型系统来做的；当然不能将它简单地集成进真正的 Chandler 代码中。另外，约翰·安德森也一直不同意采用 RDF。RDF 向来被看做理论上复杂又不切实际的典型。关于语义网的论文连篇累牍，但真正能运行的软件却不多。程序员们接连把头伸进 RDF 的世界，又接连做出类似的结论：RDF"太恐怖了"。

安德森知道编写 Chandler 的用户界面有多费劲——他干过类似的活，很清楚让一切得到控制是多么重要；这类工作常会无限期地延误。对资料库的最主要需求是方便前端开发者。而且他也知道前端开发者到底需要什么：他们需要一种"对象持久化（object persistence）"的机制。相对于将 Chandler 数据用 RDF 的主语-动词-宾语"三元"格式保存，

安德森更想把程序数据和 Python 代码一并保存为对象（“面向对象编程”的那种“对象”），这样 Chandler 程序员就能很容易地取出、操作和保存之。

现今最为普遍使用的计算机数据库——比如获得巨大成功的 Oracle 产品，或者日益流行的开源产品 MySQL——是“关系数据库”：信息被打散成离散的小单元，存储在大表中，你可以构造复杂的查询语句、有效组合这些信息——例如“展示从这个日期开始的、姓以‘c’开头、但不包括那些遗漏了街道地址的所有数据”。因为有一种叫做“事务”的模型，关系数据库通常相当可靠。事务模型记录对每个小数据的每次修改，这样，当发生崩溃或数据损坏时，还能重建信息。

实现“对象持久化”最简单的手段就是采用另一种数据库技术，即对象数据库。对象数据库把一段代码和与之相关的数据打包存储。数据和代码一起存储便于程序员编写功能代码；但也会比较慢，同时较不可靠。事有凑巧，Python 程序员们已经搭建了一个野心勃勃并且颇有知名度的对象数据库范例。它就是“ZODB”，即“Zope Object Database（Zope 对象数据库）”。麦卡斯科贴出来的那封匿名信就是在说 ZODB。当时，Python 创始人圭多·范·罗萨姆就在为生产 ZODB 的那家公司工作。

安德森下结论说，ZODB 能解决 Chandler 的问题。一开始，作为 Shimmer 的作者和 Chandler 后端部分的负责人，萨奇不同意采用 ZODB，但他承认自己的经验主要在服务器端，而在客户端——真实用户采用的程序——上则经验较少，最后终于同意了安德森的选择。

那时，罗·蒙特利和阿历克斯·托蒂克开始参加 OSAF 的会议，从完全不同的角度审视这个项目。罗提出，如果 Chandler 要满足简易 P2P 共享的需求，就得做到快速网络传输：数据必须迅速“在线上”——程序员对数据通过类似互联网这样的开放网络的路径简称——传递。他针对 Chandler 提出一种模型，将客户端及其用户界面与资料库分隔开。

Chandler 客户端使用一种互联网式的协议访问资料库——他将这种协议叫做 RAP，即资料库访问协议（Repository Access Protocol）。Chandler 通过 RAP 访问资料库，哪怕客户端程序和资料库在同一台计算机上也是如此。蒙特利坚持认为，采用这种大架构，就能解决远程访问数据的问题，而且很容易共享数据。

对卡普尔来说，解决远程访问和共享数据问题变得越来越急迫，在十月份宣告 Chandler 带来的热潮退去之后，一个由大学中的主要技术负责人组成的工作组联系到他。这个工作组叫做公共解决方案组（Common Solutions Group，CSG），成员来自二十多所教育机构，包括伯克利大学、麻省理工大学、哈佛大学、耶鲁大学和斯坦福大学。当时 CSG 正在寻找能在大学中提供价格低廉、开放、可靠的日历和邮件软件平台的解决方案。有位成员想到，Chandler 也许正合用。这简直是 OSAF 的天赐良机；不过也意味着 Chandler 的程序员们得重新设计全套用例。Chandler 最初的计划是面向个人“早期用户”和小企业用户，这些用户的特点是“以信息为中心”——他们需要处理大量邮件、约见安排和其他偶发数据。现在，开发者们还得考虑到学生和学校——有着各种不定数量移动用户的大型教育机构。

安德森对蒙特利的 RAP 构想并不感冒，不过只要这东西不妨碍他最关注的对象持久化问题，也就无谓去反对了。萨奇全然置身事外。“我有点儿缺乏自信，”后来他这样解释道。面对一群编程大牛，他想：“我知道什么啊？”

麦卡斯科负责的部分要响应的需求越来越多。他居中而坐，想找个法子叫每个人都满意。他玩杂耍的技艺高超；在他的个人首页上，曾有好些年都特别贴出在海滩上玩杂耍的相片，五个彩球在他两只手之间倒腾、画出一个拱门的形状。Chandler 的设计者们不断给出相互冲突的需求，他再灵巧也束手无策了。“OSAF 的伙计们要的东西——那整张需求

详单——不可能实现得了。”后来他这样告诉我。

哪怕对于麦卡斯科这种乐于从多种角度看问题的硬角色，眼前也是一个坎。几乎就在他加入 OSAF 的同时，他自己的生活跌入了谷底，这显然也是他陷入难以做出计划的困境的原因。他婚姻破裂、情绪低落，还患上了痛苦不堪的重复性压力损伤症[①]，不得不丢下键盘，用上了语音识别系统。假期过后，他决定改名，原来叫戴维，现在叫瑞斯（Rys）。很快他又改头换面，留了长发、染成金色，还涂上了口红。他在 blog 上写到自己中意的“不同生活方式”，不过他也警告读者别“瞎猜”：“你肯定搞错了。不过那与我无关，而且我也不怕被误解。而且，有时我觉得云里雾里的挺好玩，玩你没商量。”他显然正经历一个危险的时期——不管是什么原因，肯定无益于帮他解决 Chandler 资料库的问题。

① 因过度使用电脑导致的肌肉、关节、颈骨损害。

OSAF 的工作氛围低调而宽容，程序员们很快就接受了麦卡斯科的新名字，尽量用新名字叫他。看来，他们并不在意他长什么样，也不在意他希望别人怎么称呼自己，但他们实在想知道资料库到底会是什么样子，这样才能继续做其他部分。自从 2002 年 10 月 OSAF 进入公众的视野，卡普尔及其团队就一直在说“1 月份发布”Chandler 首个初步版本的事。到了 1 月初，显然不可能发布什么版本了，他们转而开始谈论日期未定的“首个公共版本”代码。



那是 2003 年 1 月某个阴雨连绵的星期四早晨，开发者们聚集在位于贝尔蒙特的办公室中。大会议室的白板上写满后，他们又移到堆满闲置家具的接待室。椅子不够坐，好在还有两个够大的蓝色塑料储物箱，可以当作吧凳用。留了一阵子胡须的蒙特利坐在靠后的储物箱上；在他旁边，托蒂克将头埋到笔记本电脑后面。安德森拖了把椅子坐在门口，他

头发花白，脸色红润；赫兹菲尔德两腿交叉，抬放在椅子上。

他们讨论了如何确保 Chandler 在线传递数据架构不拖后腿。安德森提出一种“预获取”的设计方案，在屏幕上显示收件箱有 25 封邮件的时候，程序应该已经准备好后面 25 封邮件，当用户向下滚动屏幕时，就能看到新的邮件信息。

蒙特利建议，应该像网页浏览器那样，当用户发出请求时就开始传递尽可能多的数据。“在网景时最普遍的做法是，”他说，“还没有得到任何数据时就要开始在屏幕上渲染页面。”在多数人还在使用低速调制解调器、笑骂 World Wide Web（万维网）是“World Wide Wait（全世界等待）”的时代，这小小手段让 Web 显得更快，也更积极地回应用户请求。蒙特利认为，Chandler 可以借鉴之。

说来说去，大家都听出来了，他不太同意一些更为基础的东西：他没跳出来说，“把 ZODB 扔进垃圾堆吧，”不过却在暗示 Chandler 并不需要 ZODB，ZODB 就是拦路虎。

安德森觉得蒙特利扯远了。“言归正传，”他说。“我们还用不用面向对象编程了？”

“用啊，”蒙特利答道。

“我们还用不用 Python 呢？”

蒙特利点了点头。托蒂克不再保持沉默，从笔记本后面抬起头来，加入了讨论，他说话带有些微贝尔格莱德口音。他说，不好说 ZODB 就是 Chandler 之所需。ZODB 其实只是在 Berkeley DB 更底层的开源数据库上做了一层抽象，用来做对象持久化。托蒂克提出，OSAF 程序员们有时间去研究和按照需要改造 ZODB，还不如直接在 Berkeley DB 之上构造自己的对象持久化代码。

安德森的“我发现了最佳方案”和托蒂克的“我们自己也能轻松做

一个”再次体现了软件复用天生的两难选择。创建还是借用？每个软件项目迟早都会到达这个岔路口。（全句的传统说法应该是：“创建、购买还是借用？”但从技术角度看，“购买”和“借用”是一样的——是同一枚硬币上商业软件和开源软件的两面。）世上到处都是别人写好的代码。拿到新项目中使用，难道不是很容易吗？那为什么还有那许多程序员扫了一眼现有的代码，就不容置疑地宣称他们自己能写出具有相同功能，而且更快、更容易、更好的代码呢？



当人们梦想把做软件变成流水线式的工作，他们常会企盼标准化的插件。新西兰学者詹姆斯·诺博尔（James Noble）和罗伯特·毕多（Robert Biddle）有时用“后现代程序员（The Postmodern Programmers）”的笔名共同写作，他们把这梦想叫做“乐高假设（Lego Hypothesis）”：“未来，程序将由可复用的部件组合而成。软件部件将在全球范围内提供。软件工程将从编程的窠臼中解放出来。”从架子上取几样零件，拼到一起，马上就能得到可用的软件——不用再痛苦地编码了！

想要走向这种编程乌托邦之路的程序员大多都发现此路不通。诺博尔和毕多的研究指出了最大的路障。他们同另外两名同事一起研究了采用面向对象技术的真实程序中的大量软件对象，发现这些构建块完全不像是乐高积木。如果软件组件像乐高积木块，那么它们就该细小、不能再分、可被替代；它们互相之间应该更为相像；它们应该“只能与有限几种相邻组件拼合”。然而，当诺博尔和毕多观察真实程序时，他们却发现，真实程序中的组件在尺寸上、功能上以及与其他组件的可拼合数量上差异甚大。它们“大小不定，就像不规则的形体，不像乐高积木。”诺博尔和毕多发现了他们称之为“普遍多样性”的现象：目力所及之处，有常者惟无常。想想看一套乐高积木，其中一些积木块只有半英寸长，

而其他积木块则长达半英里；有些用硬塑料制成，有些则是液态或气态；有些积木块藉由大家熟悉的凹凸结构相互连接，而另一些则用上了焊接、胶水或绳索。

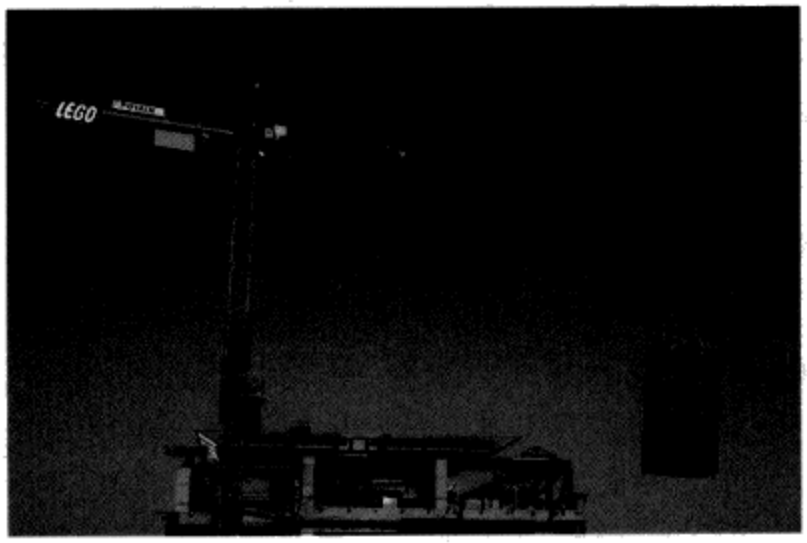


图 4-2　软件能像乐高积木一样随意组合吗？

生产出通用构造块式软件包并不容易，这显而易见。尽管屡经失败，乐高之梦仍然在现代编程史上投射下长长的影子。对于路上的每一个障碍，一代又一代程序员总能找到借力之法，避免自行开拓之苦。

可复用代码的概念可以追溯到软件的上古时代，莫瑞斯·威尔克斯（英国编程先驱，他阴郁地预言自己将终身忙于调试[①]）及同事在二战后的英国对子程序（subroutine）做了定义：所谓子程序，就是能从程序任何其他部分调用来完成某项任务的一段代码——而且，当完成任务后，还可能把结果返回给调用者。威尔克斯几乎是马上就开始往代码库中收集子程序——“这样所有用户都无须再从头写代码了”。从那时起，几乎所有的程序员都戴上了代码库管理员的帽子。

[①]见本书第 1 章

但这些例程代码库通常是孤岛一群。标准化工作往往略过这个领域；软件界有太多势不两立的标准，举目之处，四顾皆是。计算机系统中的

每一点差异——你用什么中央处理器芯片？什么操作系统的哪个版本？什么编程语言？什么数据格式？如此等等——都能惊醒乐高之梦。如多部软件工程著作的作者罗伯特·格拉斯（Robert Glass）所言，程序员们很久前就解决了“小复用”问题，通过构建子程序库来为自己减负。但一直悬而未决的问题则是“大复用”——创造并使用真正有用的软件大型可复用组件。“无关乎志向，”格拉斯写道，“亦无关乎技能。只因为难题源自软件的多样性，根深蒂固且难以解决。”

20 世纪 80 年代，一位名叫布拉德·考克斯（Brad Cox）[1]的计算机科学家迷上了藉可复用模块构造程序的问题。他设计了 C 语言的一种扩展，称作“Objective-C”，旨在改进可复用软件对象的生产。Objective-C 没流行起来，他开始从经济角度而非技术角度来看问题。1990 年，考克斯简述了一种创建和积累软件的手段，利用这种手段，程序员可以编写和销售小组件，别人购买后能集成到自己的系统中。

考克斯认为，如今我们仍处于软件的手工作坊时代；需要“将编程从单人手工活演进到有组织的企业式生产”。现在的软件开发者就像殖民地时代的枪械工人，他们钟爱手工制造每个螺丝和螺母。我们还没有取得从“枪械工人挫铁条般做软件”到“其特性能被理解和列入标准目录的可信、稳定的组件，构成商业上的强健知识库，就像成熟工程领域中的手册一样”的全行业飞跃。撇除那些复杂的技术不谈，计算机硬件行业努力完成了这种转换：只要会用螺丝刀，就能往主板上安装板卡——扩展计算机功能的电路板——而不必事先了解板卡上的芯片。同样，制造商也可以采购现成的芯片，无须操心芯片是怎么做出来的。为什么不能这样做软件呢？

考克斯做了尝试。他创办了一家叫做阶石（Stepstone）的公司，该公司致力于“向 C 语言系统搭造者提供插入式芯片级软件组件”。尽管他们取得了小小的成功，但却没有真正做起来。他后来写道：“坏消息是

[1]考克斯当时认为，软件对象应该可以类比为集成电路，基于这个理念，他发明了 Objective-C。Objective-C 也受到 Smalltalk-80 的影响，后来成为 NeXT 支持的自由软件。Apple 收购 NeXT 后，在 Mac 平台上形成了完善的 Objective-C 开发环境。

这次试验显示，即便采用最新的技术，要想设计和制造既有用又真能复用的组件、为组件写文档以便于客户理解、移植组件到潮水般不断涌现的新硬件平台上、确保最新的改进或发布版本不与现存接口冲突、将组件销售到类似威廉姆斯堡[①]枪械行业那种鼓励从头做起的价值体系，都是极其困难的。"

互联网兴起，考克斯再战难题。在 1995 年出版的《超级发布（Superdistribution）》一书中，他描述了在互联网上开辟软件组件新市场的方案。他指出，在当今的计算领域，代码被复制几乎不可追踪，但代码被使用却很容易被追踪。那为什么不编写允许自由复制、但跟踪每次使用并根据使用次数付费的软件呢？比如，我正在编写一套电子邮件程序，在编写过程中，找到一段别人写的垃圾邮件过滤代码，决定把这段代码集成到我的程序里。你不用花钱就能获得一份我的软件，不过我按月向你收取使用费，并且分一部分钱给垃圾邮件过滤代码的作者。

考克斯期望"超级发布"模式能成为可信软件组件市场的始推力，不过他的点子再次出师不利。当今基于 Web 的软件服务——用户无须下载、接触或"拥有"软件，而只是通过网页浏览器运行，免费或少量付费——最为接近考克斯梦想中软件组件的超级发布和可计量使用次数的世界。同时，程序员也走向了免费的开放源代码，不再安装那些知道自己被运行并要求付费的代码。

考克斯目前住在维吉尼亚州的马纳萨斯（Manassas），从事咨询业务。他说，尽管每天使用开源软件，并把自己看做是 Linux 黑客，但他仍然不认为开源是解决软件复用难题的现实答案。"他们确有一套经济模式，这套模式对于那些由声望而不是金钱驱动的人自有其效果，"他说，"不过更多人还在操心怎么还贷款。而我却在意如何驱动这大多数人推进复用。"

① 威廉姆斯堡（Williamsburg）位于弗吉尼亚州，美国独立战争前是英国殖民地。当地生产的毛瑟枪是英国士兵的主要武器，均产自农舍或小商店，手工制造。



不光是技术障碍和经济隔阂把开发者与他们承诺的乐高之地分开，有时人类的天性也来横插一脚。没错，如艾瑞克·瑞蒙德所写的那样，优秀程序员重写和复用。但程序员总以创作为荣，叫他们拾人牙慧，无异于拂其逆鳞。

20 世纪 90 年代热门专栏《人件集（The Peopleware Papers）》的作者拉瑞·康斯坦丁（Larry Constantine）对程序员不热衷于考克斯想法的现象做了一种实在的解释。康斯坦丁写道："不幸的是，多数程序员喜欢写程序。其中一些人喜爱编程胜过饮食和沐浴。他们中的大多数宁肯写代码，也不愿细看文档，或查询目录，或去找其他笨蛋程序员写的蠢代码……在同等条件下，程序员会选择从头设计建造，而不是重复利用。"有时，软件开发者深受"此处尚未创造"综合症的折磨，偏信于自己的技术和所在机构的力量，以至于不相信他人所造之物；有时，他们不肯花时间研究其他人的工作，甚至不肯瞥一眼别人的东西是否符合自己所需。"对于典型的程序员来说，假使要花 2 分 27 秒的时间去找一样东西，"康斯坦丁写道，"他们就会认为这玩意不存在，然后就去重新创造它。"

开源式开发和互联网这对革命双胞胎理所当然地开始改变这种习惯。Google 缩短了找东西所需的时间，即便程序员只有 2 分 27 秒的注意力，也足敷使用。像 Perl 和 Python 这样的编程语言也提供了大量代码，省得再写那些不太重要但却耗时甚多的代码，比如处理日期时间或对文本排序等。例如，截至本书写作时，Comprehensive Perl Archive Network（Perl 存档大全网络，CPAN）就收纳了来自 4359 位作者的 8107 个"模块"；每个模块都能从互联网上免费获得。

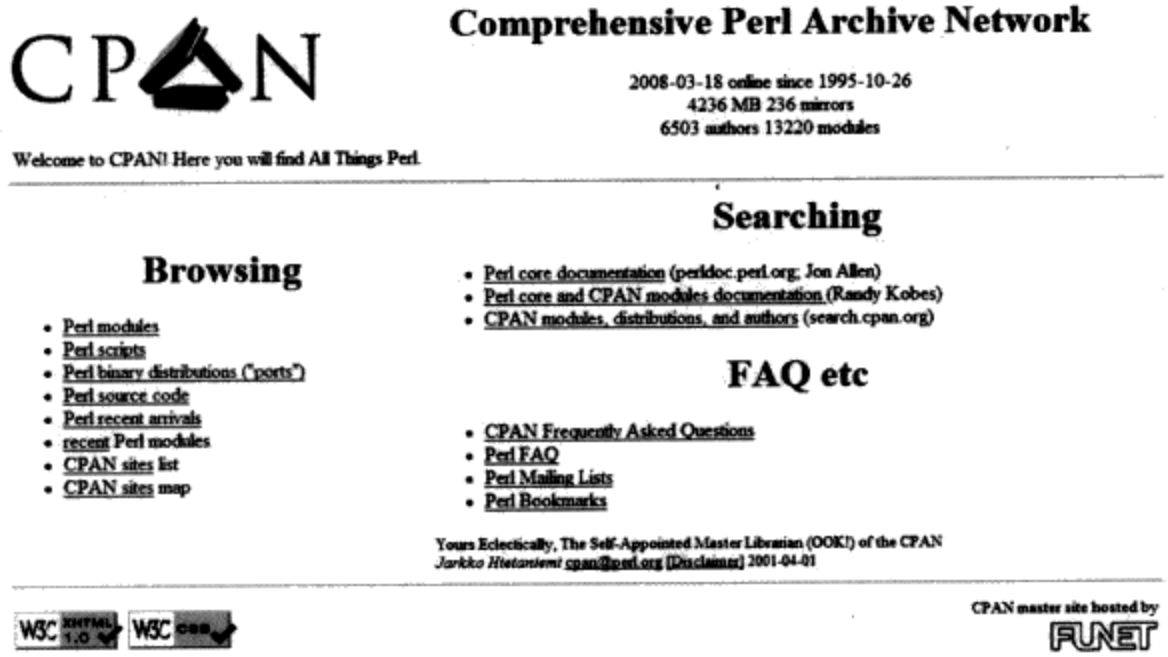


图 4-3　CPAN 网站

这些程序组件仓库可能是软件世界中最接近乐高之梦的部分了。但它却远不及最热心的贡献者们所企盼的那样有用并且被采用。在同一类组件库中，往往有许多种不同选择，每种选择还有许多不同版本。可用的代码以及每种代码包所能做的事如此之多，多到连老手们有时都会忘记有哪些可用代码；他们停止了四处找寻，转而从头开始编写某些其实是现成的功能。

Python 专家安东尼·巴克斯特拜访 OSAF 时，说到打造一种叫做 Shtoom 的互联网电话应用的想法。处理音频需要性能良好的代码；因为操作是“实时”的——必须在数据进来时立即转换，否则音频就会出现时断时续的情况，所以得快速处理得到的数据。“原始 Python 差不多就够用，”巴克斯特说，但还需要做一些改进或优化。巴克斯特给出了几种四处找来的不同解决方案：一种叫做 Psyco 的特殊 Python 编译器；一种叫做 Numeric 的数学处理器；一种叫做 Pyrex 的将 Python 编译为 C 代码

的工具。

然后他想起来：Python 本身提供的“audioop”模块就是用来高效操作音频数据的。Python 语言的标准发布版本已经具备了他想要的功能！和别人不一样，他本就该知道此事：他是 Python 最近一个版本的发布经理，在从开发、测试到最终发布的整个过程中，照看着这种编程语言。而他却全然忘记了有这种自己苦苦寻找的工具存在。

电池确已内置，如 Python 拥趸常说的那样。但有那么多的电池，谁能做到心中有数？这难题可以推而广之到每种编程语言和软件构建的每个领域。“持续跟进代码库中可利用的内容，”编程专家沃德·坎宁汉（Ward Cunningham）[①]说，“是最大的信息过载挑战。”

[①]Wiki 的发明者，也是软件模式和极限编程先驱。

Sourceforge 是 Web 上的开源仓库，容纳了超过 100,000 个项目的代码。少数项目拥有大量用户；相当部分项目仅有一人参与。你想要让计算机做的任何事，在这里都能找到相应的程序——从处理家庭录影到处理商务文件。然而，看来无论是一般而言的程序员还是特定的开源程序员，都愿意重新发明轮子。所以，如果你想在 Sourceforge 上找一种视频转换程序，就会得到 190 种可选方案。这种积习在程序员圈子中已经扎了根；从很久以前，好些程序就开始在名称上加前缀“Yet Another（又一个）”，这种命名方式表明，不管你做什么，总有许多人在之前就做过同样的事情。有 YaBB，“Yet Another Bulletin Board（又一个公告板）”，有 YASSP，“Yet Another Solaris Security Package（又一个 Solaris 安全包）”，还有 Yet Another Bittorrent Client（又一个 Bittorrent 客户端），不一而足。1994 年，当斯坦福大学的两个毕业生合伙开办了雅虎公司（Yahoo!），公司名字也聪明地缩写了“Yet Another Hierarchical Officious Oracle（又一个层级化非正式神谕）”。



对于 Chandler，米奇·卡普尔最清楚的就是，他不想做一个“Yet Another”项目——不做 Outlook 的克隆版本，不做又一种平平无奇的个人信息管理器，不做又一个什么东西。Chandler 倾力而为，无非是想求异——做一种应人所需的软件。Outlook 这样的程序将电子邮件和任务列表分开、把任务列表和约见事件分开、把约见事件和随笔记录分开；这类程序大概会给你一个小工具栏，上面画着些图标，让你能分别查看每种类型的数据，但你却无法将这些数据混配、无法将它们互相关联、无法在多个地方存储同一数据，也无法不费吹灰之力就能转换数据格式。在 Chandler 里面，所有数据——电子邮件、地址列表、约会事件、to-do 列表以及随笔记录——都被简化为“条目”，你想怎么组织和显示这些数据，都能得偿所愿。

常规程序都遵从按类型区分信息的惯例，把数据分别存储在 OSAF 开发者们所说的“地窖”中。Chandler 开发者们呐喊着：打通地窖！

这不是头一回有人喊这样的口号了。数字时代独行侠泰德·奈尔森（Ted Nelson）曾在 20 世纪 70 年代提出，计算机是“梦幻机器”和个人解放的推动力。他发明了“*超文本（hypertext）*”一词来形容内嵌链接的文本，通过超文本可以跳转到其他位置。（奈尔森认为今天的 Web 是其复杂版概念的私生子。）奈尔森还杜撰了“*intertwingularity（互融性）*”一词，凸显出“地窖式存储”所忽略的信息复杂度：“人们假装能将事物深度分层、分类及序列化，但是他们做不到，”他说，“事物之间互相交织。”

然而，尽管有千夫所指，“地窖”——只保存一类项目的独立容器——却顽强地延续下来。现代计算机的图形界面根源自真实世界——桌面、

文件、文件夹等。在真实世界中，一份文件同一时间只能在一处存在。在计算机数据的虚拟世界中，理论上，不应有这种限制。当然，在抽象层级较底层的部分，在计算机硬盘的某个物理扇区上，文件的确是存储在某个唯一位置。但只要在上面覆盖足够多的软件层，就可以通过多种渠道向用户展示这份文件，让用户可以从许多不同的位置访问到文件。用户会喜欢这种方式。

不过，如每位办公室职员知道的那样，在今天的计算机上你通常没那么大的自由度。到处都是地窖。我们的大脑知道在现实世界中物品总放在唯一的地方，所以也会首先想到这种保存方式。而在计算机的最底层，数据也的确保存在唯一的地方，按照这个思路做软件是自然而然的事。不会出错。程序员的“思维模型”——关于程序部件如何组合、数据在哪儿存放，以及用户查找或移动数据时会发生什么——向用户的思维模型看齐。

地窖遍地皆是，因为它简单、可靠且明确无误。它也反映出数字计算的二进制天性，要么 0 要么 1，要么开要么关，要么“真（true）”要么“假（false）”。要向这种系统中引入模糊状态、设计一种可以在多个地方存储同一项目的架构并非不可能之事。但需要更艰苦的工作——Chandler 团队就在干这个。

在卡普尔和 OSAF 所有人的想法中，Chandler 的资料库应该支持那种灵活适应、自由变换和非地窖式的存储，体现出 Agenda 之魂。而且它还该支持 P2P 架构设计所需的在线同步功能。ZODB 够用了吗？谁也说不准。除非是自己写的代码，否则很难确定一段代码是否真管用。

程序员们一再发现，可复用软件之梦有一个悖论：几乎总能找到一段满足大部分需要的代码。但这些拿来的代码所不能做到的部分，恰恰是项目与众不同的创新之处——也是创建这个项目的出发点。

一月会议开了几个钟头，话题围着是否使用 ZODB 转，但始终议而不决。吃过午饭，卡普尔站到白板跟前跳了几下，跑鞋的脚跟重重踏地，白色 T 恤衫的边缘爬出了腰带的束缚。他努力想从这次讨论中总结出点东西来。

“我一直在想怎么就这么难，”他一边说，一边在白板上开列原因。他问道，安德森是否太过关注程序员写软件的难度，而忽略了用户的成本。他开始列出资料库的需求：

（1）降低 Python 程序员编码难度。

（2）支持高效的在线操作。

（3）能够处理单个大数据量条目，和极大量的条目。

（4）必须可靠，采用传统的数据库“事务”手段，确保在程序崩溃时能恢复数据。

（5）支持查找和索引。

卡普尔又加了几项，然后回到列表顶端，在第一行前面添了一行：

（0）用户体验。

当天没有做出资料库的具体计划，但每个人都有活干了。约翰·安德森要和罗·蒙特利以及阿历克斯·托蒂克开会，给他们上一堂面向对象图形界面编程的应急课，告诉他们自己为什么相信“对象持久化”如此关键。凯蒂·帕兰特要准备一份文档，说明应该采用什么手段让 GUI 程序员能方便地存储数据。还有瑞斯·麦卡斯科要研究 ZODB，看看它是否能很好地支持卡普尔开列的需求。

“我注意到，”卡普尔说，“这些新任务偏离了我们原先的开发计划。我确信一路做下去总会重回正轨。”



2003 年整个冬天，开发者们都在做搭建真正 Chandler 之前的尝试性工作——努力脱离原型阶段，开始编写基础代码，即最终程序的代码雏形。安德森用 wxWidgets 和 wxPython 画出了粗糙的第一版用户界面。赫兹菲尔德做出了地址簿或联系人管理器的概览版。由于不知道程序怎么保存数据，谁也不想写太多代码；就像是还没找好地基就开始砌墙一样。具备实际功能的程序看起来如同海市蜃楼：不管开发者们商议多少次，不管他们以为自己做了多少决定，目标依然遥不可及。

部分原因在于，尽管卡普尔是 Chandler 一切事务的最终决策者，他仍然感觉自己没法左右关于资料库的争论，而且也不愿有所偏向。安德森虽然坚信和清楚自己的技术选择，却不够果断决绝，当别人提出问题，他总要再花时间去考虑。

但最主要的原因却是大家都在等瑞斯·麦卡斯科开始构建资料库。可是麦卡斯科却感到蒙特利和托蒂克把 Chandler 带入了歧途，他认为他们总在重改前议，比如关于 Python 和 ZODB 的选择。后来他说，如果当初没受这种个人情绪影响，他也许能找到更有力的技术观点来反驳他们；然而，他只是满怀沮丧地看着对方实施“阻碍策略”。而在蒙特利和托蒂克看来，麦卡斯科不能胜任，无法很快做出 OSAF 能用的东西来。蒙特利质疑对象持久化——他认为对象持久化更像是象牙塔里做的白日梦，而不是证实有用的编程技术——是否值得付出劳动。

2 月 12 日，麦卡斯科在他的 blog 上写道：“我还没为 Chandler 写一行代码。一行 Python 代码都没写。一行 C++代码都没写。我没往源代码

控制系统签入哪怕是一行代码。”

到了 1 月底，眼看项目日趋停滞，赫兹菲尔德建议——不管有没有资料库，甚至不管是否已经准备好向公众展示代码——现在是集成之前单独编写的代码、合成里程碑版本的时候了。程序员们开始往 OSAF 的 CVS 库签入代码。CVS 源代码控制系统追踪代码的每一处改动，还可以“回滚”到之前的任意版本。

就这样，2003 年 2 月 20 日，OSAF 发布了 Chandler 第一个里程碑版本。其实“版本”一词并不确切：这只是个“内部版本”，供 OSAF 开发者所用，并不是为公众开发的。这个版本尽可能限制访问。这是个“pre-alpha（alpha 之前）”版。在后端，它还得依赖萨奇的 Shimmer 原型。在界面上，它看起来像是个网页浏览器，还有前进和后退按钮。它提供了粗糙的日历和联系人管理器，因为凯蒂·帕兰特和安迪·赫兹菲尔德编写了这部分功能。

它功能很少。但至少在媒体开始关注 Chandler 后将近 4 个月，它给 OSAF 全体成员带来了些许安慰和鼓励。

在程序员们向自己发布这个 Chandler 版本那天，我在喷泉边偶遇卡普尔。“我们一直在努力！”他笑着说，“我们真的在开发软件。”



几个月前，当每个人都认为 OSAF 会加快脚步时，卡普尔答应在 4 月份 O'Reilly 的新兴技术大会（Emerging Technology Conference）上发表一个关于 Chandler 的演讲。O'Reilly 新兴技术大会关注 P2P 软件和其他新技术。现在看来，显然不可能在该次会议之前完成 Chandler 的新资料库了，不过大会还是可以被当作一种有效的期限，推动开发者们尽快做出酝酿已久的公共版本。

代码越来越多：除了帕兰特的日历和赫兹菲尔德的地址簿，产品经理林智超也利用业余时间为 Chandler 做了一个“外挂包”，或者说一个插件程序，用来阅读 blog 文章的 RSS 源。他把它叫做 Zaobao，即中文“早报”的音译。Zaobao 展示了 Chandler“可扩展性”的惊鸿一瞥。“可扩展性”的意思是说，Chandler 应该超越传统 PIM 的电子邮件、日历、任务列表、地址簿的世界，允许程序员编写扩展代码，帮助用户用非地窖式的程序组织管理各种信息。如果你用 Zaobao 读一篇文章，其中谈到某个事件，你想把这条消息转发给朋友——或者把这个事件放到日历里面，此时会发生什么事？用 Chandler 就万事大吉。不用剪切粘贴，也不用切换到其他程序。

酷！但还差得远。这个版本还包括了一些新特性：在没有新资料库支持的情况下，赫兹菲尔德采用开源的即时通讯协议 Jabber，做出了用户共享数据的初步功能。但卡普尔和其他人都觉得，Chandler 的首个公共预览版不见得会很给人留下深刻印象。在开源世界中，惯常的做法是有一点代码就放出来让所有人看到，同时吸引志愿者来提供帮助。不过 OSAF 在初次宣布 Chandler 的时候，在地上挖了个超级大坑；现在，6 个月后，开发者们暗自担心会让在邮件列表中热烈鼓励和提出建议的人们失望。

在 3 月底的一次会议上，刚加入 OSAF 负责管理编程团队的迈克尔·托伊宣布：“这个版本的目标在于展示 OSAF 的内在生命力。”

“智慧生命，”安迪·赫兹菲尔德插了句嘴。

“没错，”托伊答道，“只要它尚在成长，就不一定非做到有用不可。”

“……也不必那么有智慧，”赫兹菲尔德重复道。

1010101010101010110101101110

美国西海岸时间 2003 年 4 月 21 日下午 3:07，第一个公众版 Chandler

0.1 上载到 OSAF 服务器。24 小时内，就被下载 15000 次。

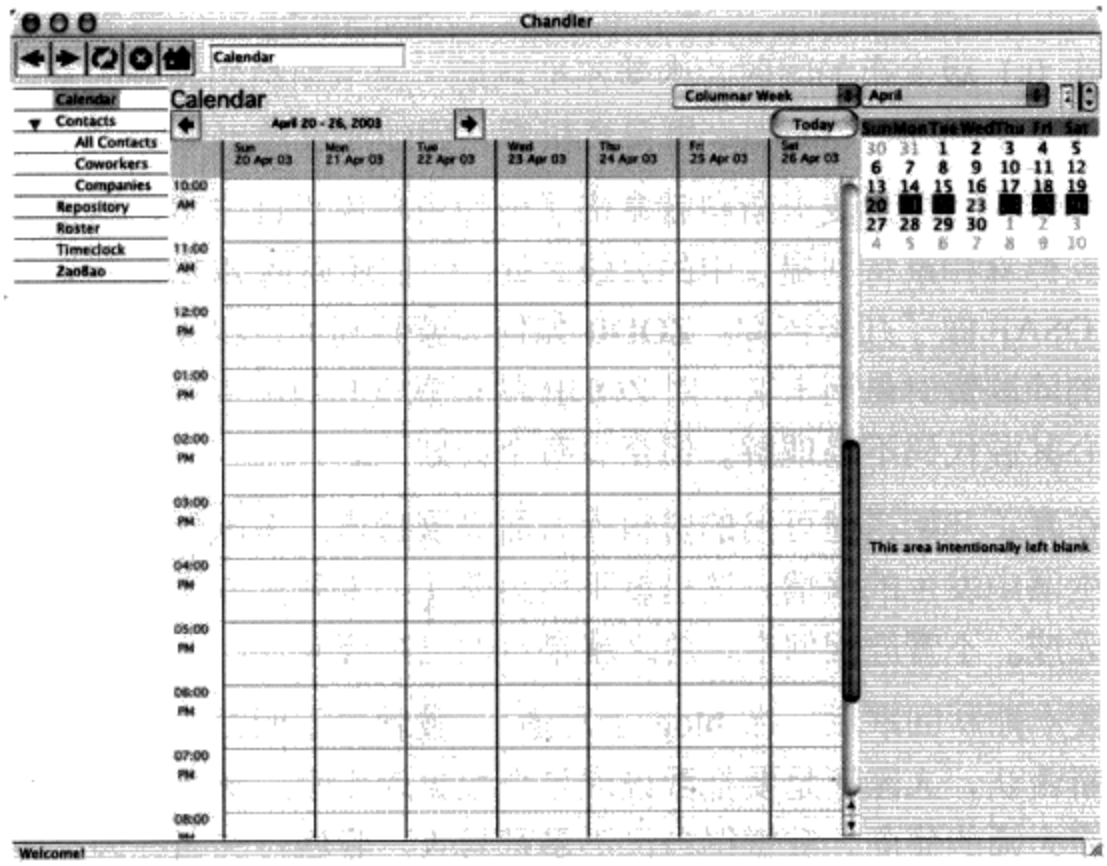


图 4-4　Chandler 0.1 版

这次比首个内部版本强一些。可以往日历里面添加约会事件，也可以在联系人列表中增加姓名和地址了。但它还是粗糙、缓慢，许多功能都不可用。启动时，欢迎窗口里的文字宣称："底层数据库仍待大改。现在用的数据库会被抛弃。也就是说"——这里用了红字——"暂且别在 Chandler 里面保存重要信息。敬请理解。"

卡普尔在 blog 上写道："我把 Chandler 0.1 叫做'B 超'版本：胎儿只能在母腹中存活，但如果仔细查看，就能看到小小的手臂和腿脚在划动，你会相信最终它将诞生。"



在 0.1 版发布的同时，卡普尔把瑞斯·麦卡斯科叫到办公室，告诉他进展并不顺利。Chandler 需要资料库，但他还没做出来。

其实在上个月麦卡斯科就拿出了一份资料库设计方案，这份方案消除了安德森和蒙特利提议中的分歧之处。他们还邀请 ZODB 的领头开发者到 OSAF 做了几天咨询。ZODB 如何与蒙特利的资料库访问协议协同工作的问题暂时悬而未决，对 ZODB 的可靠性和效率尚存疑问，还有就是担心其源代码不易理解。但至少有个计划了。

在真正开始搭建可用的资料库方面进展缓慢。考虑到这几个月麦卡斯科生活中发生的事情，这也就不足为奇了。卡普尔告诉他，他基本上一事无成。不管原因何在，总之就是没做出东西来。

麦卡斯科回家后，在 blog 上写道："我就是不明白，为什么尽管自己不断否认，人家还是要把我说成是个数据库专家。最近发现有被当成数据库大师的倾向，我觉得自己不是那号人。我做过许多在内存中移动数据以及在内存与二级存储设备间移动数据的工作，这些倒也算是典型的数据库行为。不过我绝不是数据库专家。"

可 OSAF 现在要找的正是数据库专家。



2003 年 5 月 21 日，开发者们又一次聚集在贝尔蒙特办公室的白板前，探讨资料库问题。这是旨在驯服 Chandler 各种"大蛇"的长达数周的会议中的一个。会议由迈克尔·托伊主持。桌子旁出现了一张新面孔：安蒂·维达（Andi Vajda），他于前天加入 OSAF，负责资料库开发。

"这次会议可长可短，"托伊起了个头，"我相信这些目标是 Chandler

数据存储方案所要达到的。”他在白板上写道：

（1）以程序员为使用目标，提供革命性的数据模型

（2）数据可在任何地方存储

（3）数据不能被破坏

（4）数据可被快速取得

（5）条目尺寸可以很大

蒙特利反对使用“革命性的”一词——他认为这是吹大牛。赫兹菲尔德也指出，应该用一个更普通但也能说明问题的词，这样大家都会满意。

卡普尔有点不耐烦了：“别咬文嚼字了。”

“咱们有很高的目标，”托伊接着说，“要努力去公平地估量这些目标的价值。但对列表本身不应该有异议。最可怕的是目标根本无法达到。咱们已经有一个建议方案。还有些漏洞。现在应该讨论这些漏洞，确保有补漏计划，这样，在有生之年咱们再也不会在说‘数据’这个词的时候只能翻白眼了。这条蛇是必杀目标。”

“那咱们应该在这次会议上杀掉这条蛇，”赫兹菲尔德说，“不仅只是订个杀蛇计划。”

“对，”托伊答道。

“龙身上插满了剑。”

“还有那些勇敢冲入火焰的骑士的躯体。”

卡普尔插话说：“瞧，咱们已经有一份基于瑞斯已做工作的临时计划，可以满足需要。安蒂昨天刚到，还要花点时间才能进入角色。但咱们肯定不应该从头再来。”

程序员们开始讨论与资料库相关的一系列问题：如何让资料库在网络

上高效工作——比如，当你试图载入一份大文件，例如一支 mp3 歌曲的时候？如何保证资料库安全？如何在资料库之间共享数据，或者让家里和办公室的 Chandler 保持同步（开发者们有时说“同步”，有时说“复制”）？如果想要把“Chandler 本地数据”（Chandler 自己存储的信息）和从其他程序中得到的数据——比如电子邮件或地址——混到一起又该怎么办？

托伊写满了整块白板。“咱们还没设计出复制和外部数据的方案、而且受其所困，这是头等大事。”

卡普尔疲倦地闭上眼睛。“*Mah nishtanah halailah hazeh*[①]，”他引述了经典的逾越节问题。“像他们说的，这晚与别晚有何不同？我多次踏进过那部电影的场景。都是土拨鼠日[②]。今天有何不同？”

1010101010101011010110110

答案近在咫尺。那年早春时节，安蒂·维达曾经来 OSAF 面试过，当时 Chandler 开发者们刚开始寻找负责编写电子邮件组件的人。面试结果一般，不过他们把他的名字记录在案，当需要招程序员构建资料库时，迈克尔·托伊想起来，这个人可能正合适。

维达是个固执、紧张、好眨眼睛的人，出生于一个奥地利人家庭、在法国长大，口音浓重。和 OSAF 的其他程序员一样，他从高中时代起就开始玩电脑。毕业时，他攻入了学校微机的操作系统。“基本上，”后来他告诉我，“我关于操作系统的所有认识，比如多任务、内容使用、硬盘、文件系统结构等等，都可以追溯回那次研究。大学里教的对我来说都是旧闻了。我总是想，‘嗯，原来见过这个。’”

在托伊给维达讲述 OSAF 资料库遭遇的困境时，维达的脑海中又猛然闪过“嗯，先前见过这个”的念头。他曾在一家叫做 Intellicorp 的人

①逾越节歌曲中的希伯来语问句，意为“这晚与别晚有何不同？”

②西俗每年 2 月 2 日为“土拨鼠日”。传说当天土拨鼠爬出地洞，若看见自己的影子，则冬天还将持续六个星期。电影《土拨鼠日（Groundhog Day）》（又译《偷天情缘》）讲述了一个天气预报员每天醒来都发现自己回到 2 月 2 日的故事。

工智能公司干过将近 10 年，使用该领域最受宠的 Lisp 动态编程语言。在 Intellicorp 和之后呆过的两家公司中，他都负责创建支持灵活存储和获取软件对象的“动态对象系统”。



安蒂·维达

“这正是我之所长。要是你给我一个对象系统，我就会找个地方放，因为我得找个地方放。一切都会回到 Lisp。来到 OSAF 时，我说，啊，还是这么回事，他们需要一套动态对象系统。这种事我干过好几次了……这东西不是什么高精尖研究项目，我很聪明，我能搞掂。对我来说，这简直就是轻车熟路。”

软件界中关于“牛仔程序员”的说法由来已久。所谓牛仔程序员，就是那些不守规矩、偏好独干、喜欢挑战的程序员。对于许多经理来说，牛仔程序员简直就是噩梦；而在许多程序员眼中看来，他们却是英雄。尽管安蒂 维达不完全符合牛仔程序员的标准，但他构造 Chandler 资料库的方式却无疑带有牛仔作风。他说，目标很简单：“解救这些人。”

在维达看来，5 月 21 日的“杀蛇会”，不过是快速重演了一回 OSAF 关于如何保存 Chandler 数据的长久争论。他大部分时间都在倾听，偶尔发表一下自己的意见。

“Chandler 的核心，”托伊说，“在于不分层管理数据。”

“哦，”维达打断他，“会有分层的。什么都没有的时候，你总要从某处开始。需要一个起始点。”

“从 UI 的角度看，”凯蒂·帕兰特说，“没有层级起始点。”

“但你总得从某处出发吧，”维达重申。也许只是用于内部维护，也许这对机器重要而对用户隐藏，但维达很确定一点：不采用某种类型的层级结构，就做不出稳定的数据存储系统来。

第二天，维达坐到地板上，开始阐述如何从资料库困境中走出来。他宣称，最重要的是在 Chandler 的“数据模型”——程序用来存储数据确切结构的定义——上达成一致意见。然后他才能开始搭建资料库。

接下来几个星期，维达、安德森、帕兰特和新近加入项目的志愿者布莱恩·史金勒（Brian Skinner）每天开会，形成了一些决议。他们重申说，Chandler 数据模型的核心应该是“条目（item）”。什么都可以是条目——对用户来说，可能是一封电子邮件、一条随笔记录或者一次约会事件；但它也要体现程序自身的一些机制，例如某“类”条目的定义（“电子邮件”或“随笔记录”或“约会事件”）。条目采用的面向对象方式加以组织，构成一种继承层级结构。所以，如果一组中的每一条目都被定义为某一“类”——例如，都是电子邮件——则它们将继承“电子邮件类”条目的所有特性。所有条目都应该保存在资料库中。

这种方案让 Chandler 具有“以条目为中心”的特点，它还引申出几个要点。其中之一是程序彻底从 RDF 的世界中转移出来，回到摩根·萨奇编写第一个资料库时 Chandler 的出生地。在 RDF 中，数据存储的基本单位是“属性（attribute）”；Chandler 条目可以包含许多不同属性（对于一封电子邮件，典型的属性可能是“日期”、“发件人”、“主题”、“正文”等等）。开发者们有时会说，更典型的面向对象条目与 RDF 基于属性的技术之间的差别，就像物理上的不同领域：条目如同原子；属性如同亚原子。开发者们担心，在原子级别之下的量子世界，基于 RDF 的数据存储会变得不稳定和不可预料。

决定选择“以条目为中心”的 Chandler 资料库方案，还带来另一个重要结果，即维达可以自由决定是否采用 ZODB。只要他能保证安德森和其他 Chandler 程序员可以在资料库中以条目形式保存或“持久化”他们的 Python 对象，他怎么做到都无所谓了。维达不倾向选用 ZODB；他认为应该基于 ZODB 所依赖的底层数据库 Berkeley DB 编写自己的代码，这样应该会更快、更好、更简单。这种方式非常接近罗·蒙特利和阿历

克斯·托蒂克关于 OSAF 最终采用何种方案的预言。

“我压根就没看过 ZODB，”维达后来回忆道，“时间紧张。倒是可以复用 ZODB，但得花点时间学习。我还可以复用在动态对象系统方面的经验，再做一套类似的东西。这样更快。否则，就得去拆解 ZODB，找出其运行机制，再把 Chandler 嵌进去。”

“而且，对于资料库有许多落差极大的需求——既要能共享，又要能复制。当然用 ZODB 肯定也能做到，不过如果我可以全盘控制，就能做得更快更好、更符合 Chandler 的要求。还有，ZODB 是用 Python 写的。如果以后我们想不用 Python，就会受困于此。我喜欢做灵活设计，即便需求或其他什么发生了始料未及的改变，我们也能手到擒来，说：“这也没问题。”

“那无关乎政治，甚至也不关乎‘非我发明症’，只是生存之道罢了。我们说，现在需要某种真正很快能做出来的东西。我知道如何做到。ZODB？我看不然。”

有 Chandler 数据模型的基础设计方案在手，维达万事俱备。6 月中旬，他和家人及亲戚到法国呆了一个月。回来时，他已经做好 Chandler 资料库的起初部分。跟下来几个月，他完成了“第一版”。虽然功能尚未具足，但已可供上一层 API——关键命令和数据定义，程序员需要用来编写真正保存和取出资料库数据项的 Chandler 代码——所用。



在电影《侏罗纪公园（Jurassic Park）》里面有一个短镜头，夹在暴龙发狂的场景中间。如果你不是程序员，这个镜头可能没给你留下什么深刻印象。公园中恐龙造反，小姑娘坐在控制室计算机前，尝试打开安全系统。她看了一眼屏幕，喊出声来：“是 Unix 系统，我会搞！”至少在接下来的 90 秒里面，天下太平了。

全世界的程序员观众看到这里都会欢呼——不仅是因为电影向大学

黑客和企业数据中心最喜欢的操作系统致敬，还因为他们自己也曾有过类似的“我找到了”时刻——在那些时刻，面对紧急情况或棘手难题，脑中灵光一闪，找到了熟悉、合用和自己可把握的方案。我明白问题所在。我以前做过类似的东西。我能搞掂。

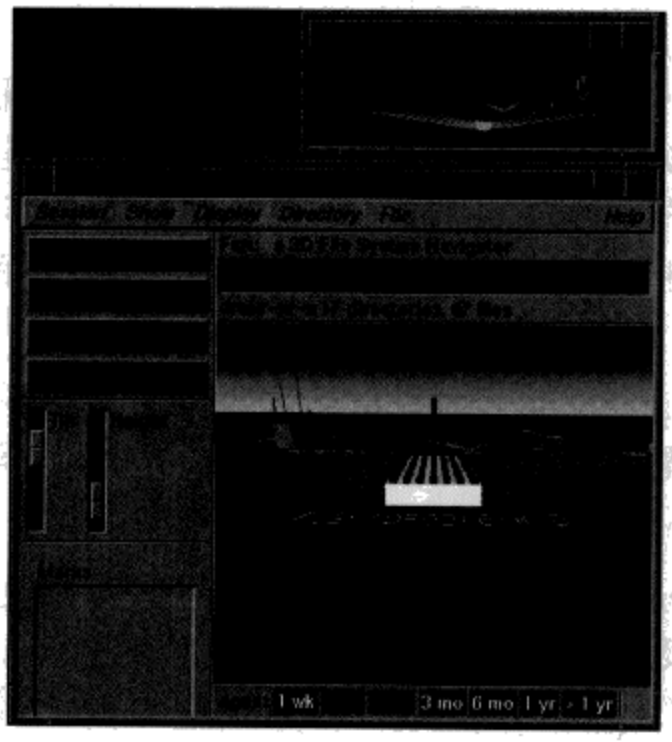


图 4-5 《侏罗纪公园》中那幕场景其实是运行在 Sun 工作站上的 FSN 三维浏览软件

安蒂·维达来到 OSAF，四处一看，一时间也有此感受，便开始编写代码。他快刀斩乱麻，终结了关于 ZODB 长达数月的争论。乐高积木式的插件不适合。他确信自己能更快更好地新做一套。然后他开始干，而且做到了。虽然还有些脆弱，但也足以鼓舞人心。它将 Chandler 团队拉出软件时间的无风带，回到有所进展的世界。

OSAF 的其他开发者还需要一点时间来了解维达的信心是否可靠。但至少他们开始有了资料库的雏形。至少，他们不再原地踏步了。

第 5 章　管束奇客和狗

[2003 年 4 月～8 月]

1. WQE

2003 年夏天的一个早晨，这些字符出现在我的电脑屏幕上。我没输入这些字符——是那条名叫 Chandler 的狗干的。当时我正在 OSAF 开会，用电脑做记录。Chandler 跑进来，用嘴拱了拱键盘。我吓了一跳，不过没太吃惊。米奇·卡普尔有两只拉布拉多狮子狗，Chandler 就是其中之一，它们获准在办公室里自由活动。



图 5-1　拉布拉多狮子狗 Chandler

Chandler 和 Cosmo 都是浅褐色的拉布拉多狮子狗，天性好动，它们是这办公室兽栏里的王者。还有白色的狮子狗 Siggy，举止轻佻，总在寻求仰慕者；Petunia 是一只英国斗犬，走路一摇一摆、满地乱嗅，老是守护在主人的办公桌旁；Porter 是一只巧克力色的拉布拉多犬，它不停四处巡游；此外还有至少 12 只轮番出现的宠物狗。2003 年 7 月 OSAF 从贝尔蒙特搬到沃德街的办公室，这里“对狗友善”。令犬主和分享办公室空间的企业们都很开心。少数没养狗的人也并不总会感到不爽。

当时，豢养拉布拉多狮子狗——拉布拉多犬和纯种狮子狗的混血种——正是潮流之选。它们是好宠物：“和其他狗类融洽相处，”有文章这样描写拉布拉多狮子狗，“非常聪明，快活而友善。能快速学会不常见或特殊的技能。活跃，有时显得滑稽。如果管束不严就会戏弄主人。”

Chandler 缺乏管束？嗯，有时它会在地毯上留下长达数码的湿迹。有天下午，它选中办公室一角作小便之所；没多久，楼下招聘服务公司的老板发现有液体滴到她的桌子上。Chandler 和 Cosmo 时不时地在办公室里相互追逐，在 OSAF 开发者可能正碰头讨论程序架构要点的大会议室里跑进跑出。如果会议室门关上了，它们就猛烈撞击双扇门，就像攻城的大木槌一般。

多数时候，有这些狗在，还是挺有意思的。它们嬉闹着，沉浸在无视周围人类满脑子协议、接口和数据模型的幸福之中，让这地方温馨起来。但随着 OSAF 成员日多、需要更多办公空间时，放任人狗混处就有问题了。需要建立某种管理秩序。OSAF 组建了一个“养狗兴趣小组”特别委员会，由人力资源经理牵头，处理这类事务。

显然这是桩交易：狗主人们，包括老板在内，都想要让狗陪在身边。程序员们则需要集中精神。不过也有些程序员喜欢让狗陪着，而有些则觉得备受干扰。对狗友善的工作场所固然好，但也需要狗主人们负责管束他们的宠物。

养狗兴趣小组承诺每两周开一次例会，设置“狗频道”匿名电子邮件地址，接受对狗的投诉。还正式在办公区域中划出了几块“无狗地带”。

交易不好做。但怎么可能不做困难的选择就成事呢？



混血狗 Chandler 从这个初创软件项目得名，它那张热情生动的脸也出现在 Chandler 程序每个后续版本的欢迎窗口中，以威廉姆·维格曼（William Wegman）[①]式的风格呈现出来：戴安全帽的 Chandler（警示——建筑工地）、穿研究室工作服的 Chandler（“现可试用”）、日历上 Chandler 的脸庞（日历功能可用了！）。这只狗自然而然成了该软件项目的吉祥物。

[①]美国摄影家，以拍摄拟人化的犬类（特别是魏玛猎犬）而闻名。

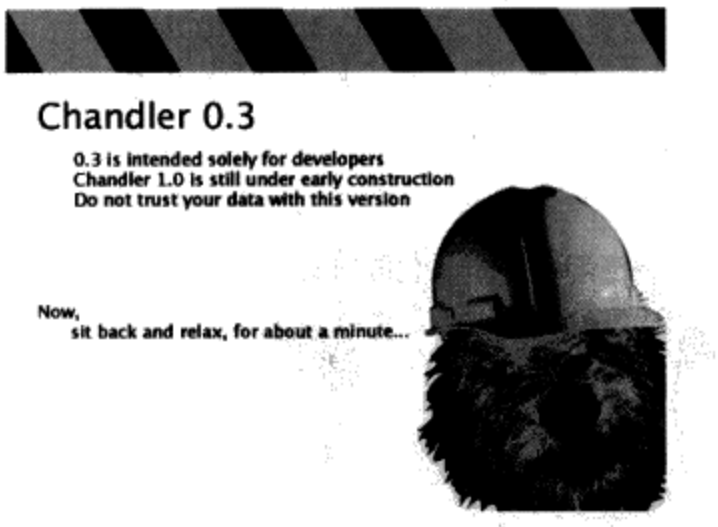


图 5-2 戴安全帽的 Chandler

*能快速学会不常见或特殊的技能。如果管束不严则会戏弄主人。*这是说拉布拉多狮子狗还是说程序员？用“经理”替换“主人”，相似度之高令人震惊。当人们用动物术语来谈论管理程序员的问题时，通常不会比作训狗，而是比作“管理猫群”。常言道，程序员是顽固的独行侠。老

调重弹不足以促其成事；他们需要哄逗和抚爱，要有耗子可以追杀，要有缺陷可以荡涤。

管理犬只和管理软件项目的相同点是必须做交易。当你选定一个首要目标——一只训练有素的小狗？一位按时完成的程序员？——就会发现自己得牺牲一些别的东西。在软件世界中，多数选择都归结为令人神伤的三向交易。乐天派称之为“质量三角”；悲观派则叫它做“不可能的三角”。不管怎么命名，通常总是坏消息。

我从资深软件开发人员和作家丹·沙弗尔（**Dan Shafer**）口中听到这条法则，早期他加入 **Salon** 团队，力图让我们这帮雄心勃勃但失之天真的报人们跟上技术的脚步。我们谈到 **Table Talk**——一个还未等软件准备好就仓促开工的在线会议系统——进展非常缓慢，令人抓耳挠腮。他往后仰靠到椅背，摸着大肚子，说道：“老话说得好：快速、便宜、优质，任选其二吧。”

在任何一个工程领域做久了，都会遭遇这令人痛苦的方程式。但在软件世界中，质量三角变化多端，成本难以测算、总是感到时间不够，体现得淋漓尽致。就像商业领域中其他坏消息一样，经理们也常以为自己的组织能在例外之列。他们以为可以三者兼得：既好、又快、还便宜。

程序员喜欢做方阵式比较；下面的方阵列出了在质量三角中可能的组合选择：

快速	便宜	优质	
☐	☐		速溶柠檬茶！
☐		☐	昂贵的赛车
	☐	☐	强壮的海龟

上列方阵只展示了一种二选一的图景，即一系列单纯的二元选择。实践中，经理们面对的却更像是一具滑钮之间互相关联的混音台。你决定多花点钱，把“成本”滑钮从“低廉”移向“昂贵”？你会发现，这会让“质量”滑钮从“低劣”一端向“优良”一端稍稍移动。如果你还想加快速度，那就得继续抬高“成本”滑钮——或者将“质量”滑钮推回去。

如果项目的实施能精确反映在成本、进度、特性或质量上的决策，则项目经理只需要坐到混音台前，调整控制滑钮就好。然而，管理软件开发项目的方式往往更为原始——要从无到有创建秩序，要回答类似“从哪儿入手”和“怎么才知道何时完工”之类的问题，还得协调难以控制的团队力量，使其朝向共同目标一致用力。



一天，OSAF 有几条狗身上抹了治跳蚤药膏。主人们在狗背上贴了小纸条，写着“正在药杀跳蚤——别抱我！”的字样。

卡普尔走过时，看了一下纸条，微笑起来。“要是人也这么做，那该多好啊！”他开玩笑说，“胸前挂上牌子，上书‘我今天很生气。’‘我不爽。’肯定能剔除人际关系中模模糊糊的部分。我举双手赞成这么干！”



2002 年 3 月份关于 Chandler 后端的争论开始后不久，卡普尔认为需要聘请一位项目经理了。要有人制订进度。要有人拍板做决定——而且言出必行。要有人创建或从别处采纳并修改流程规则。“我们已经太大了，”他在 blog 上写道，“大到再也不能只借助‘瓦肯心灵融合术（Vulcan mind-meld）[①]’来工作。”

[①] 瓦肯是系列剧《Star Trek（星际迷航）》中瓦肯星上的类人生物。他们通过心灵融合仪式达到心灵相通的目的。

卡普尔不想自己干——也不认为自己能干得特别好。作为老资格程序员和软件架构师，约翰·安德森部分扮演了这个角色。不过他不太有把握能管好那些细节任务——制订进度、推动程序员按进度工作、决定先干什么再干什么。他思考多于实干，或许不是承担这类职责的上上之选。总之，他手上总有做出技术决策和编写代码之类的活要干。

2002 年末圣诞节放假期间，卡普尔在 blog 上贴出一篇题为《设计决策：一些原则（Making Design Decisions: Some Principles）》的文章。文中第一条原则就是："必须依序执行。决定推迟某些特性，项目才成其为项目。所有事同时开干并非明智之选。"

这"所有事"可太多了。作为项目内容首次公开的一部分，OSAF 网站上贴出的特性摘要包括了 5 大类 36 种特性。仅电子邮件类就有 17 种特性。有些特性——例如"通过全文索引快速搜索"、"文内直接查看附件"以及"用户自定义视图、规则和过滤器"——虽然实现起来并不简单，但却是每种电子邮件程序的标准功能。其他特性——例如"支持用户脚本"、"在多个文件夹中归档消息"和"邮件透明加密和验证"——则是较为复杂的创举；以前没人做过，即便做过也没能成功发布。

卡普尔和 OSAF 宣称 Chandler 将允许用户通过 P2P 模式方便灵活地共享信息，无须依赖服务器就能让不同地点运行的程序之间同步数据（例如，家里和公司的计算机之间）。但 Chandler 也需要保证用户重要信息的安全。

能二者兼得吗？

卡普尔和 OSAF 承诺，Chandler 最终将会是一种全功能的个人信息管理器，一种能处理电子邮件、日历、地址簿、待办事项和随意贴的 PIM 软件。同时它也将会是一个"可扩展开发者平台"，程序员可以扩充 Chandler 的功能，让它能管理所有类型的信息，包括音乐、相片、blog 文章，以及天底下一切事物。

能二者兼得吗?

Chandler 开发者们将采用开放源代码模式，迭代构建程序并向外部公开。但卡普尔也想打造界面好、运行快、感觉佳的程序。

能二者兼得吗?

举目四望，尽是选择。

在新项目经理到岗前，项目没法摆脱这些选择。也许要几个星期，甚至要几个月。“在过渡期里，我们不能一味地等待白马骑士，”在给团队成员的备忘录中，卡普尔这样写道。他也准备好亲自做其中一些选择。当资料库之争延续时，他已经做了一个重大抉择：先实现 Chandler 的基本 PIM 功能，比如电子邮件和日历，而把开放给程序员扩展功能、管理各类信息的“开发者平台”特性延后实现。这颇令人失望——毕竟是这些很像 Agenda 的特性让 Chandler 充满激情、鼓舞了在线社区。但他们还是得等。相对于试图同时做所有事然后不可避免地失败，OSAF 更愿意集中力量打造运行顺畅的 PIM 软件——就像卡普尔在其 blog 文章《设计决策》中写到的那样，是一个包括大把“杀手级特性”、“彻底优于”竞争对手的产品，“值得一用”。

然而即便收窄到只剩一些杀手级特性，OSAF 仍然要面对大量选择，且有可能迷失在其中。卡普尔贴出了“候选杀手级特性”列表：“共享；强固、透明的安全措施；代理架构；Agenda 般的灵活性；[在此提出你的建议]。”最后的留空是对现实的让步——Chandler 的规划还有漫漫长路要走——在有新的好点子冒出来时，可以灵活应对、抓住机会。但同时它也向延误敞开了大门。



为 OSAF 寻找软件开发经理的工作进展缓慢。尽管居此位者手握大

权，却许多程序员却不愿为之。有些程序员学足“呆伯特（Dilbert）[①]”连环漫画中——里面处处针对那位留双尖发的老板（pointy-haired boss），偏爱缩略语的程序员们也用 PHB 来指称这个角色——全面抗拒的做法，视任此职者为厌物。（有位程序员在某个开发者网站上留言说，“经理就是我的痛。”）另一些人认同良好管理的价值，但却不够胆色担此重任。

寻找项目经理的工作仍在进行中，罗·蒙特利和阿历克斯·托蒂克推荐了一位他们在网景公司时的同事。迈克尔·托伊曾在 SGI 公司做部门管理工作。当 SGI 创始人吉姆·克拉克（Jim Clark）决定新开一家后来叫做网景的公司时，迈克尔·托伊跟他一起离开了 SGI。在那个重新定义软件开发标准的年代，他带领公司的开发队伍在浏览器战争中打了好几次硬仗，不断刷新更快发布新版本的记录。他拥有衡量好软件开发经理的唯一资质：有据可查。



双尖发老板

托伊生长于加州的利福摩尔（Livermore），从在加州大学圣克鲁兹（Santa Cruz）和伯克利分校读书时就开始编程，当时他编写了早期的龙与地下城经典游戏 Rogue。起初他只是为学校的 Unix 微机编写 Rogue，没想到，几年后这个游戏居然如此流行。20 世纪 80 年代早期及中期，继 IBM PC 之后，麦金塔计算机横扫业界。托伊和几个朋友开了家公司，把 Rogue 游戏移植到新系统上。



图 5-3　迈克尔·托伊写的 Rogue 游戏

[①]著名 IT 讽刺漫画，里面有一位自以为是的中层经理，头发梳作魔鬼撒旦式的两个尖角，是主要反面人物。

“我在高中时就编写了自己的第一个程序，从那以后，对世上其他一切都失去了兴趣，”他回忆说，“解决难题、解释给计算机听、看到计算机照足你说的做，这个过程真的很让人觉得有所回报。离开大学后，感觉就像——别的程序员也常这么说——傻瓜们**付钱给我**做那些如果他们自己也懂我就收不到钱的活。”他停顿了一下，面无表情，“直到今天还是这样。”

网景公司的早期雇员都获利颇丰，但这还不足以维系团队。当公司明显被微软（以及自己的错误）扼住喉咙时，托伊离开了，暂时脱离软件工业。2003 年春天，他还没想清楚是否回归，但 OSAF 提供了符合他梦想中工作的每一项要求：“我喜欢做给上百万人用、在每个人计算机上运行的软件。我想浏览器的失败在于它总说处理信息最有趣的方式就是阅读……然而你一旦拥有浏览器，下一步就是组织和沟通信息，而浏览器却对此视而不见，即便这是大势所趋。这就是我从 1995 年起想要解决的问题。但我却不知道匆忙做出个蹩脚软件的压力所在。如果时间更为充裕，应该能做出更像样的东西吧？而且最好是开放源代码的。”

托伊警告卡普尔说，自己不擅长处理人员方面的事务——指导、绩效考核等等。卡普尔告诉他不用担心，他们可以一起处理这些事。

托伊于 2003 年 3 月到岗。他与那种典型的双尖发老板完全是两个极端。他长发及背、扎个猪尾巴辫，举止间流露出一种低沉的谦虚。在个人网站首页上，有一幅将他绘成《绿野仙踪（The Wizard of Oz）》里稻草人形象的图画。开会时，他偶尔会打断手下人的发言，要求他们“再说一遍给我这小脑袋瓜听吧”或“让迈克尔更聪明点”。他在 blog 上反省自己的自由派基督教信仰，而到 OSAF 后，则开始写关于管理程序员的思考。他在 OSAF 的 blog 命名为 Blogotomy，网页上写着这句铭文：“从程序员转做经理常被说成是做了‘前脑叶白质切除手术[①]’。这里就是

[①]这种脑部手术曾被认为是治疗精神疾病的最佳手段，但副作用是患者会丧失感情、不知爱恨悲喜。

一个现在很幸福的转行者的记录。”

他还为一家叫做 Undignified Labs 的公司——为基督教音乐家提供录音服务的工作室——维护网站，在这个网站上，他贴出了以下关于自己的介绍：“如果你想找人刷墙，迈克尔是你最后才会想起来的人，但却是第一个你想请来拆墙的人。”

OSAF 既不刷墙、也不拆墙；它还处于浇灌地基的阶段。托伊要做的第一件事就是制定开发计划。4 月份的 O'Reilly 大会定死了 Chandler 0.1 版的发布日程；在那之后，开发组却只做了一份很长的特性列表，同时走向了最后发布 1.0 版的漫长旅程。

卡普尔决定采用洛杉矶不同地区的名字来给 Chandler 的不同版本命名，瑞蒙德·钱德勒小说中的神秘故事就发生在这些地方。（为什么不用钱德勒小说的名字来命名呢？因为如果这样做的话，就会止步于用《长眠（The Big Sleep）》那本小说命名的版本了。）所以，Chandler 1.0——面向个人用户的第一个全功能版本——将被命名为 Canoga。提供给 OSAF 在大学里的新搭档们的第一个版本叫做 Westwood。而更长远的 Chandler 2.0 版本代号为 Pasadena，它允许其他公司为中小型企业开发定制版本。

OSAF 承诺于 2003 年底推出 Chandler 1.0 版，即 Canoga 版本。但直到当年春天，OSAF 甚至还没奠定软件后端的基础架构。对于卡普尔的程序员们，如托伊在 blog 上写的那样，就像“我们正站在一块还在冷却凝固过程中的地面上。”怎样才能按时完成呢？

托伊知道，做法只有一个：制订进度计划并尽力跟上。



“管理，”商业哲人彼得·德鲁克（Peter Drucker）[1]写道，“全与人

[1]当今最著名的商业管理大师。

有关。管理的任务就是让人能协同工作，益其长处、弱其短处。”我们惯于把管理看做是闪耀着科学光环的商业学校技术应用：生产力统一衡量，投资利润率。德鲁克的定义初听起来经不得考验——他好像是在说管弦乐队指挥或是舞台剧导演。德鲁克强调管理艺术胜于管理科学，对人的关注胜于对定量的关注。德鲁克发明了“*知识工作者（knowledge worker）*”的说法，并深入研究其内涵，创造了不可估量的价值。他想提醒我们，数字只是管理的一个起始点，并非终极目标。

在软件项目管理中存在一种充满讽刺意味的天性，尽管程序员工作时所用的计算机和编程语言都要求精确缜密，但软件编写工作却奇怪地对绩效考量有抵抗力。数十年以来，程序经理们尽力寻找一种准确的方法来测量该领域的生产力。程序员每天的工作成果是代码，而软件生产力最明显的量尺也是代码行。然而这量尺却不能令人满意，有时甚至具有欺骗性。诺博尔和毕多在研究可复用软件对象时发现，代码行各有不同。在代码量和程序完成度、质量以及对用户的价值之间，并无可靠的关联关系。

安迪·赫兹菲尔德说了一个关于他早期在苹果公司时的导师比尔·艾金森（Bill Atkinson）的故事。比尔·艾金森是传奇般的软件创新家，他创造了 Quickdraw 和 Hypercard。艾金森当时负责苹果 Lisa 计算机（麦金塔计算机前身）的图形界面。Lisa 的开发经理制定了一套管理规则，要求工程师们填写每周编写的代码行数，艾金森怒不可遏。“他认为用代码行来衡量软件生产力太愚蠢，”赫兹菲尔德在回忆麦金塔岁月的著作《硅谷革命（Revolution in the Vally）》中写道，“他认为自己的目标是编写尽可能又小又快的程序，用代码行数做判断标准只会鼓励程序员写出臃肿、蹩脚的代码。”

在首次被要求填写新管理表格的那个星期，艾金森刚重写完 Quickdraw 的部分代码，使之更为高效、快速。新版本比旧版本少了 2000

行代码。该怎么写报告呢？他填了个负数：-2000。

如果说以代码行数计算不太可靠，那么，衡量软件生产力的其他通用方法也同样不可靠。你可以跟踪程序特性或“功能点”——但它们很难被整齐划分成难度或尺寸相似的单元，只能以对每个特性完成时间的主观预测告终。你可以跟踪缺陷数量，用降低缺陷总数的过程来衡量生产力。找出缺陷虽然让你更接近完成目标，但有时却引发新一轮的缺陷暴增。而且修复缺陷的时间长短各异，所以你的团队可能花几个星期就能把列表上的缺陷数量从 100 减少到 12，却发现最后这 12 个缺陷坚如磐石，只见时光飞逝，仍旧毫无进展。另外，不管你采用哪种不可靠的度量方式，都会面临弗里德里希·布鲁克斯在《人月神话》中提到的那令人不爽的事实：不同程序员的生产力相距甚远，常会达到 10 倍差距。所以，考虑如何配置项目人员，和预估项目所需时间一样叫人充满挫败感。

多数软件经理都很清楚这些困难，也不会临时抱佛脚。他们会先列出要做些什么，再细分为一系列任务，也有办法跟踪任务的完成情况。软件经理充分意识到软件时间的悖论和危险性，他们希望每个程序员都预估——或至少 SWAG（盲估）一下——剩下的任务还需要多少时间才能完成。然后经理把这些信息集中到一起，估计项目的完成时间，并且——如果像经常发生的那样，评估结果是软件在下一个千禧年到来之前也不可能做完——开始做出选择。



非正式的技术项目管理方式在惠普的“闲逛式管理”概念中达到巅峰，汤姆·彼得斯（Tom Peters）的《追求卓越（In Search of Excellence）》一书普及了这种概念。这种严密的方法要求经理们走出办公室，遍访坐

在隔间里的下属，和他们谈话。“做得怎样了？”这句话听起来不太像最锐利或最具分析性的询问，但通过闲逛（或散步）进行管理确实极具革命性：与其盯着表格看，不如走出去自己观察雇员们真正在做什么。它鼓励人们观察和接触他人。

但 MBWA——缩略语掮客们很快就用这个词来代替“闲逛式管理（management by wandering around）”——却没能很成功地移植到软件领域：开发进度是闲逛的管理人员看不到的。没人比瓦茨·汉弗里（Watts Humphrey）对这种困难描述得更准确了。瓦茨·汉弗里是软件管理大牛。20 世纪 60 年代，在弗里德里希·布鲁克斯离职后，他负责领导 IBM 软件团队，之后又在卡耐基-梅隆（Carnegie-Mellon）大学创办软件工程学院（Software Engineering Institute），提出了一整套令人眼花缭乱的软件开发方法论。

“对于制造业、军队和传统的硬件开发，”汉弗里写道，“管理者可以巡视商店、战场或实验室，看看大家都在做什么。如果有人做错或没有产出，管理者看上几分钟就能知道。然而，对于一组软件开发者，你却不能单凭随便看看就知道他们在做什么。你得询问，或者仔细检查成品。要足够警醒和有学识的经理才能够说出软件开发者在做什么。假使你让他们做点别的，或是采用一种新的实现方法，很难说他们真会照办。”

所有这些盲点都意味着“经理不知道项目进展情况”。难道经理就不能直接问团队成员吗？“多数开发者都乐意告诉经理自己的进度，”汉弗里这么说，“问题是，就目前的软件实践而言，开发者们对于自己的进度也不比经理知道得多。”



汉弗里的看法令人气馁，但至少实实在在：你怎么能度量看不到的

东西？如果每个成员都不知道自己的进度，你又怎么能判断整个团队的进度？

今天的商业世界中，在对软件项目表现普遍不满意的表象背后，有着更深层的焦虑——只怕问题的根源不在于管理技术的失败，而在于做这类工作的人的天性。对于许多管理者、甚至对于销售或其他部门的同事而言，程序员根本就属于另一类物种。和他们沟通相当困难、令人丧气。他们对常用的奖励/惩罚刺激手段毫无反应。他们是奇客（geek），他们正是问题所在。

大约 10 年前，“*奇客*”成为通用词，用来描述那些与计算机沟通易于与人类沟通的人。之前常用的“*nerd（讨客）*”一词，中学小圈子和防笔漏保护袋的味道太浓。“奇客”的词源可以追溯到几个世纪以前（其中一种写法甚至可以在莎士比亚[Shakespeare]的《第十二夜[Twelfth Night]》和《辛白林[Cymbeline]》中看到）；原义“傻子”或“笨蛋”定型于 20 世纪早期，来自于马戏团的杂耍表演，观众可以看到“奇客”演员吃昆虫或是一口咬下活鸡的脑袋。20 世纪 90 年代，当个人计算机踏上美国商业和文化舞台的中心，那些需要联网或安装打印机但又不懂技术的人，只能寻求身边“计算机奇客”的帮助——至少没有用这个词的字面意思。“奇客”一词，就像“*queer（同性恋）*”和其他用于边缘人群的标记性词语一样，最初具有贬义，到后来就被对象人群广泛接受并引以为荣。

从《新黑客词典》（Jargon File/New Hacker's Dictionary[①]）各个版本的更替中，你可以回溯“奇客”一词的演变。《新黑客词典》是程序员行话和花絮的集大成者，目前由《大教堂与集市》的作者艾瑞克·瑞蒙德维护。早期版本将奇客定义为“以（计算机）程序缺陷为食——不善社交、身有恶臭、面色苍白的偏执狂，具有奶酪刨丝器一般的人格特点。”稍新一些的版本中的定义则带有更多自尊的意味：“奇客：专注于己事的人；追求技术（特别是专业技术）和梦想、不融入主流社会的人。”

[①]最初由斯坦福大学的拉斐尔·芬克尔（Raphael Finkel）于 1975 年编撰。原始版本见：http://www.dourish.com/goodies/jargon.html。艾瑞克版本见：http://www.catb.org/~esr/jargon/。



奇客

当这个词走向褒义，还出现了词组 *geek out（奇拗）*。在 2005 年《纽约时报》社论版上，尼尔·史蒂芬森（Neal Stephenson）解释道："奇拗，意思就是沉浸到细节中直至变态境界——而且乐此不疲。"真奇客有本事奇拗于任何事物——甚至清洁厨房。有位名叫默林·曼恩（Merlin Mann）的生产力软件狂徒在 2004 年开通了名为 43 Folders 的 blog，很快就在程序员中间传开了。他在 blog 中为一本题为《让家更舒适（Home Comforts）》的书大唱赞歌："看了书中**整 9 页**关于怎么洗盘子的内容，我那四脚蛇脑子里有些疯狂的皮层沟回突然展开了……我想，我那极度整洁（也有无限耐心）的女朋友可能错以为我只对那些充斥图表文档的有趣系统感兴趣。"

并非所有奇客都是程序员，反之亦非所有程序员都是奇客，但两者之间的重叠非常彻底，人们也自然将他们等量齐观。在程序员中，奇客又按不同维度两两分类：Linux 奇客和 Mac 奇客，标准奇客和脚本奇客，数据库奇客和加密奇客，每种都有自己的亚文化参考点。奇客的社会性表现也各有不同，按部就班型工程师喜欢削尖铅笔、与他人循定例相处；而无拘无束型黑客则留着与地心引力对抗的发型，本能地反抗一切秩序。

他们所共有的是对特殊知识的狂热和糟糕的人际关系——要么拙口笨舌，要么唇枪舌剑，如果沟通对象缺乏这种狂热的话。你说不知道怎样配置家庭网络？有些奇客会很高兴地帮你搞掂，然后给你演示 6 种优化手段，另一些则没头没脑甩下一句"是 RTFM 的问题，老兄！"就转身而去，你只好在 Google 上查那**该死的手册**。

只要程序员还躲在小黑屋里做着他们的神秘活计，难以交谈或问而不答倒也无关紧要。但当计算机进入每间课堂、每个家庭，当每家公司都开始围绕定制软件重塑业务流程，不闻不问的奇客管理方式就越来越不可容忍。程序员们发现自己被冠以"首席信息官"之衔，参加会议、对媒体发言。与外界交流的确痛苦，但独善其身已不复可行。

为什么奇客和非奇客不能和睦相处？最近国防采办大学（Defense Acquisition University）一项题为“信息技术团队中的人类变数（The Human Dynamics of Information Technology Teams）”的研究试图深入了解来自77个团队的600名IT专业人士（多数是程序员和他们的经理）。研究者采用6种评估手段，包括观察、自我汇报以及一系列调查，发现IT专业人士的个性特征与普通人群完全不同：

超过四分之三（77%）的样本偏爱考虑后做决策，而只有23%的样本偏爱凭感觉决策。前者比例明显高于普通人群，普通人群两类决策方式比例大约对等。他们所说的“思考者”通常喜欢逻辑、客观、非个人化的决策方式，全力关注因果关系和保持客观（问题第一，人类第二）的清晰度。

41%的受访IT专业人士认为自己是内闭型思考者（即内向自闭又喜欢思考），比例两倍于普通人群。内闭型思考者常喜单枪匹马，避开团队和协同工作方式以及任何支持这种方式的培训。这类群体最不易与人相处，可能还会避免与同事之间建立开诚布公、相互信任的关系。

多数程序员的共事者都了解到，负责编码的同事很像是理性的独行侠，他们对此并不太吃惊。“好些人觉得，与IT专业人士沟通，要比与死人沟通还稍稍难一些，”研究项目领导人波茨·艾伦·汉密尔顿（Booz Allen Hamilton）的分析师同事艾比·麦克尼斯（Abby Mackness）在向一群国防承包商雇员和顾问介绍研究成果时这样开玩笑说。

在最极端的情况下，程序员的行为特征——避免眼光接触、很难读懂身体语言、沉迷于技术奥秘——很像是患了艾斯伯格综合症（Asperger's Syndrome）。艾斯伯格症被定义为轻度自闭症，和严重自闭

症患者不同，艾斯伯格症患者往往拥有高智商，也能控制自己的行为。从 20 世纪 90 年代开始，一直在上升的艾斯伯格症和自闭症的发病率在硅谷地区陡然跃升。史蒂夫·希尔波曼（Steve Silberman）在 2001 年一期《连线（Wired）》杂志上撰写了题为《奇客症候群（The Geek Syndrome）》的文章，讨论这种科学家们相信是遗传因素作怪的情形为什么在程序员的后代里如此集中。希尔波曼写道：

笨手笨脚、在真实世界中易受挫折，孤独的心灵沉浸在数学、符号和代码的虚拟世界中……

硅谷是自然形成的社群，充满热情和才智的人们从世界各处移居此地，他们致力于让聪明的机器更聪明。终日与比特为伍的辛劳工作要求他们有一颗自闭症患者般的心。然而，建成这块飞地的代价，却暗含于近 10 年来几乎每项关于自闭症的基因研究当中。研究者们一再断定自闭症基因不仅可能遗传自传统的自闭症患者，还可能遗传自那些只有少许典型症状的人……眼前发生的事是第一个证据，说明赋予那些轻度自闭症患者特殊才能的基因——让他们成为梦想家和我们技术未来的架构者——也能在他们下一代身上撒播病患的种子，这种可能性令人遍体生寒。

无论自闭症发病率上升的原因是什么——是硅谷奇客之间通婚产子的大规模试验使然，还是该归咎于互不相干的各种因素——在自闭症状与奇客特征之间总有着某种交叉。有些编程人群的观察者说，为了与要指令的机器沟通更密切，许多程序员切断了与人性一面的联系。而艾斯伯格症/自闭症研究者则认为，更可能的原因是程序员对自己进行了编程，让自己能像听人说话一样听到机器的发声频率，甚至听得更清楚。

这有助于他们写出更有效率的代码、设计更有效率的算法。坏处是他们不懂得如何打造能完成人类用户设定目标的程序。当遇到与非奇客交流这种简单需求时，他们也会陷于困境。



几乎就在弗里德里希·布鲁克斯撰写《人月神话》的同时，该领域的另一位程序员杰拉德·温伯格（Gerald Weinberg）也写了一本题为《计算机编程心理学（The Psychology of Computer Programming）》[①]的书。这是使用人类学和心理学工具研究这个快速发展学科的首次尝试。温伯格发现，程序员尽管不合群，却**真的需要与他人倾谈**——形式越随意越好。

[①]又译作《程序开发心理学》。

温伯格讲述了某大学计算中心发生的事情。由于聚集在自动售卖机旁边的学生太吵闹，官僚们决定搬走售卖机。不这样做怎么行。在售卖机被移走的日子里，原本没太多事可做的中心的辅导人员忙个不停。等待辅导的学生一直排到了门外。除了售卖机被搬走外，并没有其他变化——学生数量、课程数量、作业数量都维持原状。最后，中心主任让一位社会学毕业生去问那些排队等待辅导的学生怎么回事。当时的发现印证了今天商业学者们称之为“冷水机效应（water-cooler effect）”的理论：

> 当学生到达计算中心时，典型行为是先取得程序输出，然后走到咖啡机旁，一边喝咖啡，一边看程序，还把程序展示给周围的兄弟们看。由于学生的多数问题都类似，所以极有可能在咖啡机旁边就能找到知道问题所在的人。通过这种非正式的组织形式，绕过了正式的咨询机制，所以辅导人员的工作量也就有效降低了。

温伯格指出，要点是“非正式机制总是存在，而且如果没真正理解就改变它是很危险的，要避免扰乱原本运行顺畅而且你无法以同等代价替换的系统。”自温伯格发此宏论之后的数十年中，管理者们从粗暴地破坏这类非正式机制转向笨拙地尝试鼓励这种机制。设计办公室布局的人开始有目的地规划出推动温伯格著作里那些大学官僚欲图以秩序之名压制的喧闹声的空间。与此同时，程序员们已远远走在建筑师和经理们的前头：他们发明了一系列技术好让彼此保持联系，他们通过各种新的团队协同工具拓展了软件领域。

这些工具有时被统称为“群件（groupware）”，包括即时通信、聊天室、缺陷跟踪、源代码版本控制和传统的邮件列表工具等等。当 OSAF 开始工作时，blog 和 wiki 这两种较新的概念正崭露头角。每种类型的软件都能向不同渠道输送信息：比如，每当有人签入新的代码补丁时，源代码控制系统就能向一个邮件列表发消息。在你自己意识到之前，你的每个工作日已经被几十种这类渠道传送的信息吃了个一干二净。

在早年的编程领域，团队在纸上共享知识，使用叫做项目工作簿的活页夹处理复杂的协同更新流程。布鲁克斯的《人月神话》花了大量篇幅描写这些规程。然而，当程序员能够扔开纸面作业时，他们就这么做了。最早的群件是道格·恩格巴特的 NLS，这套产生于 20 世纪 60 年代的软件成为后来个人计算机领域无数革命性进步的先祖。每一代新的群件都重现了恩格巴特关于“提靴带”的理念——使用计算机工具来提升开发计算机工具的效率。

对 OSAF 而言，该现象的镜像逻辑生动而痛苦地展现出来：当 OSAF 的程序员们苦苦纠缠于各种不同的协同工作方式时，设若他们突然意识到假使只有 Chandler 可用反而会更便于打造 Chandler，一定会陷入悔恨之中。

有观察者尝言，程序员寻找替代与人直接沟通的方式的热望，源自

于他们不耐枯坐一室、和人面对面交谈（face-to-face，FTF）。典型的奇客无疑不喜欢那些切入和侵占编码时间的管理会议。实际上这种隐士般古板的程序员相对而言只是少数。程序员往往喜欢讨论本行之事。各类群件的流行不是因为程序员避免与其他程序员见面会谈，而是因为他们无法不讨论。今天，同一机构的程序员常常散布于不同时区；开源项目的程序员也许永不会聚于一堂（“人肉空间[meatspace][①]”）。为了完成工作，他们得有办法随时询问和得到解答、在团队内部传播答案，并保存答案以备不时之需。

[①]指真实的人类社会，相对于cyberspace（虚拟空间）而言。

1010101010101011010110110

当沃德·坎宁汉创造出今天被称之为 wiki 的东西时，他并没有想要发明什么不得了的玩意儿；他只是想让工作更轻松些罢了。20 世纪 90 年代初期，作为一位资深程序员和面向对象编程神秘艺术的追随者，沃德·坎宁汉很早就投身于模式语言运动——在软件开发中实现建筑哲学家克里斯多夫·亚历山大（Christopher Alexander）的理念。亚历山大著有《模式语言（A Pattern Language）》[②]一书，籍由观察成功建筑物的共同因素或模式，发展出一套描述建筑的语法。软件模式语言爱好者们致力于在编程领域做同样的事情。20 世纪 90 年代中期，当 Web 骤然流行，他们召开了一次大会，坎宁汉受命创建一种程序员可以用来在网上共享软件模式的超文本资料库。

[②]又译《建筑模式语言》。

十年后，他回忆了第一个 wiki 程序诞生的过程：

我设想，嗯，先运行这个资料库，收到你发来的文本格式的模式描述，运行一段小脚本，把文本转换为 HTML（编写网页的语言）。我制定了一些简单规则，例如，空行表示新段落之类。不过发来文件的人们

连这么简单的规则都不能遵守，这让我很吃惊。处理了三份模式文档后，我已经很不耐烦了，就做了个提交文档的表单。

人们提交文档后，还要劳烦坎宁汉修改页面。所以他在每页下端添加一个按钮，上写“编辑本页”。不需要密码。任何人都可以点击按钮后开始编辑页面。“太简陋了，有点不好意思给人看，”他这样说过，“这么简单的东西真能运转起来吗？”

坎宁汉加了点东西：如果一串单词内部用了大写字母——即所谓骆驼命名法（CamelCase，例如 ItMakesWordsLookLikeThis）——软件就会在数据库中搜索这些单词。如果有页面以该串单词命名，就自动链接到该页面；如果没有，则自动创建空页面模板，等人来填充内容。而且，由于 wiki 软件保存了每个页面的所有版本——跟踪全部修改，很像程序员用的源代码控制系统——就可以在发现有人胡乱编辑时轻易地回滚到之前的状态。

坎宁汉的第一个 wiki 应用波特兰模式资料库（Portland Pattern Repository）在十年间积累了大约 3000 个页面。它推动了整个 wiki 浪潮。（wiki 一词来自坎宁汉在旅行中听到的夏威夷语，意为“快点”。）它最为著名的后代是由志愿者搭建的维基百科（Wikipedia），在许多贡献者的增补修订之下，维基百科成为互联网上最知名和最常用的资料库。如今，小型公司都争相将 wiki 手段引入业务建设中。多年来企业都在投入构造内部网——让员工共享信息的内部网站——但在官僚作风和复杂设计的重压之下这种努力常会轰然倒塌。看来 wiki 提供了一种 Web 协同的快餐式手段。



当迈克尔·托伊到岗之时，Chandler 项目组正开始苦恼于沟通渠道

过载的问题。2002 年 10 月，卡普尔向外界公开 OSAF 时，在一个内部邮件列表之外，还开放了 4 个公开邮件列表。卡普尔和另外几个人已经开始写 blog。开发者们在 IRC（Internet Relay Chat）聊天室里讨论；每天的讨论都会存档以备查找。程序员们使用 CVS 源代码控制系统保存代码，签入修改过的代码而不至打断他人工作。他们采用 Bugzilla 跟踪缺陷和修正记录。第二年 1 月，OSAF 还开放了一个公开的 wiki，用来共享便签、研究成果和想法。

基于实践和世界观两方面的考虑，卡普尔都坚信透明即善。他解释说，OSAF 的工作进展理应开放给公众审查，因为 Chandler 本来就是开源项目，还因为开放就是好。程序员和经理当博采众议，不应闭门造车。这没有传统软件开发实践所暗示的那么困难。最终，鼓励开放总比强调秩序与连贯性好一些。

几乎从 Chandler 打开 wiki 之门当天起，抱怨就没断过。有些 OSAF 开发者认为在文本框中输入文字的界面设计得很傻，拒绝在工作中使用 wiki。还有人过度使用，导致每日便签很快便堆积成山。所有人都认同 wiki 的“随时随地加页面”机制会导致混乱：新手不知从何处入手，而惯用者则很难找到之前访问过的资料。

驯服 wiki 的重任交给了程序员和电子邮件管理专家，绰号“鸭子”的凯特琳·舍伍德（Kaitlin “Ducky” Sherwood）。凯特琳加入 OSAF，帮助实现 Chandler 的电子邮件功能，并管理项目文档（帮助程序员和用户理解程序架构和特性的材料）。舍伍德尝试从不同方向解决这个难题——把 wiki 切分成数个区块：Chandler 区块保存官方文档，Journal 区块保存会议记录和程序员日志，还有一个叫做 Jungle 的“沙箱”区块，用户可以在这里随便乱改。Chandler 团队从编程领域借用一个术语来指称重新构造 wiki 的过程，他们把这个过程叫做“wiki 重构”。对于程序员来说，“重构”意味着重写某段代码，使之更简洁、更清晰、更易于阅读，且不



凯特琳·舍伍德绰号“鸭子”

影响其功能。重构常被比之于园艺修剪：它永远做不完。

自舍伍德在 2003 年中对 Chandler wiki 做了大改之后一年，开发者们又有了同样的抱怨：怎么才知道哪些页面提供最新信息、哪些页面已经过时？在 wiki 上怎么才能看到项目的概观？目录在哪儿？如果说 blog 和邮件列表过于依赖其以时间排序的结构，那么 wiki 的随意格式则是把用户扔到了海里，四围茫茫、漫无头绪。



尽管 OSAF 使用了多种工具，作为开发经理，托伊发现这些工具并没有给出最重要问题的答案：上周每个程序员都完成了哪些工作，下一步应该做什么？有些开发者在 wiki 页上贴出便签，有些偶尔向邮件列表发消息，有些在 blog 上贴出进度。可项目的整体状态却无处可寻。

托伊可不想再增加一个工具来解决问题。在网景公司，他们也曾采用 Bugzilla 来跟踪缺陷之外的其他事务。它是专门用于跟踪缺陷的，所以并非一般任务管理的最佳工具，不过他认为，用 Bugzilla 总比再让团队成员们再多用一样工具好。

有些程序员对 Bugzilla 很满意。有些程序员觉得在网页表单上输入任务令人抓狂。约翰·安德森拒不执行；他抱怨说，得要花上一整天时间往 Bugzilla 里面输入任务列表，什么代码也写不了。而对于那些需要汇报每日进展的 OSAF 非程序员们，Bugzilla 特别不友好。

OSAF 采用 Bugzilla 作为官方项目管理工具长达 6 个月，由于积极响应者寥寥，摩根·萨奇着手为团队开发一套状态管理器（Status Manager）——一种基于 Web 的工具，改进了输入任务和依人员、依项

目、依时间和依状态排序查看的过程。2003 年 11 月，萨奇最终发布了这个新工具，在开发者中引起极大震动。如果他们还不能使用 Chandler 来管理自己的工作，状态管理器也算是个不错的权益之计。

同样，萨奇用来编写状态管理器的数周时间——以及“鸭子”舍伍德用来改进 wiki 的数周时间——都没有用来打造 Chandler。



在《人月神话》中，布鲁克斯称巴别塔为“人类第二项大工程”（第一项是诺亚方舟）和“第一次工程大惨败”。布鲁克斯称，巴别塔的管理者们拥有现代管理顾问渴求的所有“成功先决条件”，包括清晰的任务目标、充足的人力、材料和时间，以及足敷使用的技术水平。巴别塔的失败是因为建设者们“无法讨论；所以也无法协作。当协作失败时，工作便陷于停滞。”

在软件管理中，协作不是马后炮，也不是无足轻重之事：它是工作的核心，决定采用何种工具和方法有可能成就或毁掉项目。管理这些工具容易诱使项目偏离正轨。当“我们自己做！”的呼声高涨，奇客们总是欢欣鼓舞。麻省理工大学有一句著名的（也可能是假冒的）涂鸦：“我宁肯写帮我写程序的程序，也不写能写程序的程序。”亚伯拉罕·林肯（Abraham Lincoln）也有类似的名言：“给我 6 个小时砍树，我会用前 4 个小时磨斧头。”这条原则被写入商业顾问手册《高效人士的 7 个习惯（The Seven Habits of Highly Effective People）》[①]，激起了每个程序员制造工具的热情。但如果执着于此，管理最佳的项目也会一头载到阴沟里。Java 语言发明者、编程大师詹姆斯·高斯林（James Gosling）在 blog 上探讨这种偏好，他写道：

[①]史蒂芬·R·考维（Stephen R Covey）所著畅销书，又译作《积极人生》。

有两个暗面：人们谈论最多的是切实磨快斧头然后回去砍树。但多数时候打造工具比作手边工作要有趣许多。

对我来说，“磨斧头”的最大问题是循环往复，有点像是芝诺悖论（Zeno's Paradoxes）[①]：你先花上三分之二的时间完成制造工具的任务。但制造工具的任务本身也需要工具：要磨斧头，得有块磨刀石。于是你再花三分之二磨斧头的时间来做一块好磨刀石。在打造磨刀石之前你还需要花时间找合适的石头。所以你得去据说是盛产优质磨刀石的巴芬岛（Baffin Island）[②]北部海岸地区。要去到那边，你得先造一具狗拉雪橇……

解决芝诺悖论不成问题，因为虽然工作量看似无限，但指数衰减值也是无限的。而在软件开发中则没那么幸运。打造工具的事越来越多，很快就让你忘记刚开始要干什么了。

程序员们还用“*给牦牛剃毛（yak-shaving）*”来形容这种状况——艾瑞克·瑞蒙德的《新黑客词典》中定义 yak-shaving 是一种“看似毫无意义的举动，但确乎是解决某一问题的必要工作，解决此问题就能解决其他问题，如是循环递归，最终解决工作中的真正问题。”

程序员总将磨斧头或剃牦牛毛之类的个别行为判断为“确乎必要”。当然有些是必要的。对于许多软件经理来说，很难判定磨斧头和剃牦牛毛在何时偏离了项目的原始目标，也很难让乐此不疲于打造工具的程序员们重回正轨。



OSAF 在 4 月份发布了 Chandler 0.1 版，5 月份的大部分时间都在开“杀蛇会”。迈克尔·托伊面临的紧迫任务是如何制订 0.2 版的发布进度。

[①]亚里士多德《物理论》中转述古希腊数学家芝诺提出的四个悖论，证明运动的不可分性。分别是：运动物永远不能到达目的地；阿基里斯永远追不上先出发的乌龟；飞矢不动；一半时间等于一倍时间。

[②]加拿大第一大岛，也是全世界第五大岛屿。

是等到编码结束、运行正常后再挑选合适的特性、宣告发布呢？还是定个死日子，然后不管做到什么程度，就在那个日期发布做完的部分，宣称 0.2 版开发完成？托伊本能地选择了后一种方案——“时钟驱动版本”或“标定日期的里程碑”。无论如何，这只是个 alpha 版产品：没人在用，也没人用得起来。如果不定个“最后屏障”日期并且坚持之，就会冒让 Chandler 0.1 开发工作继续敷衍下去的风险。

托伊和卡普尔最后同意 OSAF 应该遵循时钟驱动进度，每个季度发布一次新版本。那时已是 6 月；预计 9 月要发布 Chandler 0.2 版。安蒂·维达加入后，资料库终于渐露峥嵘。程序员们工具齐备，斧头已经磨好了。

现在 Chandler 缺少的就是完整的设计方案了。

第 6 章　搞掂设计方案

[2003 年 7 月～11 月]

在为本书做调研的后期，我越来越清楚地了解到导致软件系统走向失败的各种可能性，有天晚上，我告诉自己："别犯傻。赶紧备份。"

于是我设计了一套备份方案。第一步是把劳动果实收集到一个名为"Book Project"的大文件夹里面。其中放置的文件来自我办公室和家里的计算机、所有网页地址、采访笔记和午夜草草记下的只言片语。

就在手边！但 Book Project 文件夹自身却深埋于树状文件夹结构的第四层。不太方便。我有好一阵子得在这个文件夹上工作，所以想把它移到更方便的位置。我打开一个资源管理器窗口——微软 Windows 中用于文件导航的基础工具——找到 Book Project 文件夹。我点击这个文件夹，按住鼠标键，把它拖到——

嗯，该拖到哪儿呢？我还没想清楚。这其实很重要。可选位置太多。我要先想想！

我将鼠标指针移到屏幕外面、松开按键——根据运行另外一些程序

的经验，我以为文件夹会像皮筋一样弹回原来的位置。

其实不然。编写这个事件处理代码的Windows程序员们可没这么想。他们以为，如果你把文件夹拖到屏幕外面，就是想让它彻底消失。

Book Project 文件夹没了。全都没了。被删掉了。

我目瞪口呆看着屏幕。怎么能**这样**?

嗯，别着急，检查一下回收站——肯定在里面。只要拖出来就可以了。

那儿也**没有**。

Book Project 文件夹不复存在。它成了过去时。

我心跳加速。我心神不定。我的大脑开始思考起这个讽刺性事件——所有关于软件开发的研究都毁在一个软件错误手里了！而我身体的其余部分却正处于肾上腺素陡增的混乱境况。

我开始在网上疯狂寻找反删除工具——以前不需要这个。突然我住手了。我想起来，要想恢复被删除的文件，最好别再用这台计算机，因为这可能会在硬盘上写东西：新文件可能会覆盖你想要恢复的旧文件的残余部分。

最后，我平静下来，脑海里响起一个声音：**撤销（Undo）**。

对。当然。撤销。我看向资源管理器窗口的“编辑”菜单。就在那儿——我的救主。我点了一下“撤销”菜单项，Book Project 文件夹回来了，完好无缺。

我静坐桌前，向编写撤销功能的 Windows 程序员们默默致谢——尽管他们可能也是制造这起灾难的罪魁祸首。然后就洗洗睡了。

```
10101010101010101101011010110
```

创建“终端用户应用软件（end-user application software）”——仅人

类使用的软件——意味着将遭遇无数种人类行为与机器反馈的组合。计算机屏幕上有几十万个像素；用户可能随时点击其中随意一处，而程序必须知道应做出怎样的反应。那些疯狂用户的举动，就连最好的程序员也难以全部想到。

各种各样的约定，例如单击与双击的区别、“焦点（focus）”——屏幕上位于前台的活动部分（例如，在典型的电子邮件程序里，一开始焦点可能在位于屏幕中央的新邮件列表上，但如果你点击旁边的文件夹列表，焦点就会转移到那个屏幕元素上，它也会接受鼠标点击、键盘输入等事件）——的概念，都是为了帮忙驯服和管理这片可能性的汪洋大海。但这些只是约定，而不是自然法则；并无天条可依。它们随一代又一代软件潮生潮灭。就像生物进化一样，旧系统的残片有时也会在新系统中延续下来。

经历过命令行界面时代的用户都知道，在你敲击“回车”键之前，计算机不会“读取”你键入的内容。这已是条件反射——甚至不经大脑。大约十年前，当我向父母介绍万维网时，我告诉他们可以在浏览器顶端的地址栏输入网页地址。他们照做无误，但却毫无反应。是我忘了告诉他们，“输入完毕后要按回车键。”网页浏览器是一种鼠标加窗口的GUI 程序，而地址栏则是命令行时代的残留物，承袭了一系列旧习惯和旧行为。

通常是由程序员负责猜测程序该如何应对用户输入和机器状态的上千种可能组合。但他们却并不善于站在用户的立场考虑问题，想出合理之策。他们花费大量时间纠缠于数字化细节，他们被调教得按自己做出的系统一般地运作。他们视之为理所当然的概念，对于非程序员而言纯然是怪异之举；他们也多半不太了解用户的想法。（程序员常依赖一种称作“妈妈测试”的手段，以对计算机一无所知的父母为假想用例，有时甚至请这类用户亲自体验。）

除此之外，程序员们还要负责考虑各种极端情况和不太可能出现的情形。当保存工作成果时断电怎么办？如果用户把计算机时钟设定为公元 2099 年，日历程序又会如何？（微软 Windows 允许你这么干——当成百万人使用 Windows 这样的软件时，迟早会有人穷尽软件给出的所有操作可能性。）程序员们把这叫做边缘案例（edge case），里面常常会藏匿着缺陷。程序员们经过训练要巨细靡遗通盘考虑，他们太执着于警惕会出问题的情形，结果却难免会迟钝不灵。他们在边缘案例上绞尽脑汁，以至于偏离了中心点。

假使程序员们不是做出那些软件程序设计决策的最佳人选，谁又是最佳人选呢？



米奇·卡普尔对软件设计的细节总是抱有热情，从泛泛而谈的哲学问题层面到平淡无奇的布局细节（让应用程序显得对用户友好）均是如此。在将近 20 年里，他一直致力于将软件设计确立为独立的专业领域。

1990 年，在广聚计算机工业智者的 PC Forum 大会上，卡普尔首次发表了《软件设计宣言》一文。

没人为可怜的用户说话。在这个问题上大家信守缄默的密约……揭开表层掩饰，你会发现人们羞于承认自己发现那些设备难以使用。他们以为错在自己……我认识的每个人（包括我自己在内）每周至少有那么一次直想把那恼人的机器扔出窗去……软件缺乏可用性、拙劣的程序设计，这些都是计算机工业不足为外人道的耻辱。应该做点什么呢？计算机专业人员应当承担起创建良好用户体验的责任。也许最重要的概念进步就是意识到在计算机产品中设计的紧迫性，并将它与编程相提并论。

而在计算机业里最重要的社会进步则是为软件设计师留出位置，鼓励对用户体验的关注。

卡普尔认为，软件设计不仅只是在程序员代码之上覆盖一层诱人的图形。它是一种设想用户需求并在软件结构中满足这些需求的创造性基础工作。当架构师为团队绘制出蓝图，软件设计师就应画出平面图和正视图，好让软件工程师们据此炮制。卡普尔建议在软件领域中应用古罗马建筑理论家维楚维斯（Vitruvius）[①]关于良好设计的原则：*坚固*——良好的结构、没有缺陷；*适用*——“程序应符合其设定目标之所需”；*愉悦*——“使用程序的体验应令人愉快”。

卡普尔宣言中的好些说法今天看来足以自证，但他关于设计师应主导创建软件的断言至今仍有争议。有些程序员看不起设计和设计师，坚信自己才是程序界面和操作方式的最佳仲裁者。另一些程序员推崇代码优先，认为凭空想象软件的外形和工作方式不足为信，故“设计优先”（又称 Big Design Up Front，BDUF）也不堪用，最好是尽快写出可以运行的程序，再根据用户反馈改进它。

卡普尔从来也没有主张过设计必须独立于或先于编码。“在设计方案付诸实施的过程中，新信息会不断产生，很多时候这些新信息将导致对原有设计的修改，”他在宣言中写道，“假使设计方案与实现过程被隔离在两个密封舱中，反倒是一剂救治灾难的良方，因为这样就规避了精炼和改进的自然过程。”

卡普尔自己一直就是 OSAF 的主导设计师，但每个人都有自己的设计理念。赫兹菲尔德对可用性的关注近乎偏执。和他交换意见时，通常总会引到类似“用户不会要那个”或“该让用户决定”之类的说法。

但直至 2003 年夏天，OSAF 仍然没有做出 Chandler 的完整设计方案；

① Marcus Vitruvius Pollio，罗马建筑师、工程师，名著《建筑十书》的作者，估计他在凯撒时代（大约公元前 1 世纪）已活跃于建筑界。

程序仍处于有一堆原则、热情和许诺的阶段，还没有画出蓝图。部分原因是由于卡普尔及其团队在招聘设计师时遇到困难。他们需要一位图形感强、理解界面设计、而且能和程序员们紧密协作的人。卡普尔的标准实在严苛。他们还没找到合适的人。

再者，要确定 Chandler 的设计方案也很困难，方案确定的话也就解决了程序宏愿与技术现实之间的艰难交易问题。定不下设计方案，就难以果断做出技术选择，但如果缺乏技术路线图，设计方案也很难定下来。如此总归结到鸡与蛋难题的境况，在 OSAF 团队中弥漫了一种沮丧的氛围。

网景小子罗·蒙特利和阿历克斯·托蒂克对缓慢的进展深感失望，离开了自己为之志愿工作达六个月之久的项目。后来蒙特利把他的离开归咎于性格不合："我太具破坏性。我想推动事情尽快动起来。在整个组织中，别人更想要谨慎、缓慢的软件开发方式，我却感觉不对路。"托蒂克转投另一新项目，改造日益流行的开源开发工具包 Eclipse，使之能用于编写 Python 代码。蒙特利则搬回太浩湖，从事创投工作。



2003 年 7 月，安蒂·维达开始为 Chandler 资料库挖地基。米奇·卡普尔与约翰·安德森、安迪·赫兹菲尔德组成了一个非正式的工作组，后来迈克尔·托伊也加入进来，一起制订 Chandler 用户界面的搭建计划。在 Chandler 单个片断功能上的工作，例如凯蒂·帕兰特的日历功能部分，已经停下来。程序缺少关键部分，缺少一个在不同时期被叫做"大表"、"概要表"或"超级挂件"的结构——能以摘要或概略形式呈现电子邮件收件箱或任务列表或日历事件的部件。

得有人迈步前进、开始编写超级挂件。赫兹菲尔德曾为 Vista 雏形写过类似的界面，但他认为，对于真正的 Chandler 程序，应该由安德森负

责这个部分。安德森同意了。不过在确信自己明白卡普尔希望超级挂件怎样工作、且项目主要贡献者们一致同意如何正确地构造它之前，他可不想一头扎到代码里。对于怎么做出来，他也有自己的想法。



“问题是，”在一连串长会的其中一个会议上，米奇·卡普尔问道，“我们该怎样调度工作，免得没完没了徒然浪费光阴而不见成效？”

“只有陷入循环才会没完没了，”赫兹菲尔德答道。



这段对话开始之时，卡普尔、安德森和赫兹菲尔德决定，用户和程序员都要基于他们一开始叫做“Chandler 文档”的东西工作。和文字处理器文档一样，Chandler 文档也保存数据；但它更像是电子表格文档，也会是一种模板，保存用于处理数据的代码和规则。而且，如同奠定了卡普尔事业基础的软件创新——电子表格一样，Chandler 文档也要把原来只有高手程序员能做到的功能——扩展程序的功能和用途的能力——直接交到普通用户手上。

主题转向关注文档之后，该小组的讨论开始被叫做文档架构会议。这些会议贯穿了整个夏天，有时一周要开上三、四次耗时一下午的会。他们的目标是解决开放性问题，让安德森可以开始编写超级挂件，但他们发现很难缩小讨论范围。对人们如何使用 Chandler 完成工作的设想，变成了关于代理（agent）的讨论。代理是安迪·赫兹菲尔德长久以来追寻的愿景，它是一种能代替用户处理常规任务的自动化、半自主性软件例程。关于用户应该拥有多少微调功能和 Chandler 的界面布局应该如何

的争论，很快变成围绕移动设备上的 Chandler 或网页上的 Chandler 之类话题的论战。而且似乎每次讨论都回到那个问题——看似已经解决实则仍会跳出来咬一口——即 Chandler 或许可以采用 Mozilla 的一些部件。

自安德森初次检视并拒绝采用 Mozilla 工具已将近一年时间，1998 年网景公司开放源代码后，这些工具从 Netscape 浏览器软件中分离出来。自此之后，随着原网景公司正式雇员——从托蒂克、蒙特利到托伊——的加入，“我们是否该再看看 Netscape？”这个问题在 OSAF 时常回响。而把这个问题再次摆到到台面上的原因，则是由于米奇·卡普尔当上了 Mozilla 基金会主席。

以前，Mozilla 的核心程序员们名列网景公司花名册上，1998 年美国在线（America Online）收购网景公司后，他们变成了 AOL 员工。同样原因，2003 年，他们又成了时代华纳（Time Warner）员工。在互联网爆发的顶点，时代华纳与 AOL 合并，之后又萌生悔意。当 IT 行业陷入最低谷，AOL 的管理者们问道：请再告诉我一次，为什么要发工资给那票只会向外传播代码的程序员？Mozilla 的未来茫然难测。

卡普尔曾通过米歇尔·贝克（Mitchell Baker）与 Mozilla 有过联系，贝克自 2002 年来为 OSAF 兼职工作。作为网景公司的老兵，贝克还在 Netscape 浏览器的开源进程中扮演了核心角色：她撰写了 AOL 发布 Netscape 代码的许可文档。随着 AOL 对 Mozilla 的态度渐趋冷淡，米歇尔·贝克也于 2001 年被解雇。不过她还是以志愿者身份继续帮助领导 Mozilla 项目。卡普尔请她加入 OSAF，以她在开源方面的经验帮助 OSAF 搭建 Chandler 周边开发者社区。卡普尔也从她那里了解到 Mozilla 在 AOL 内部的不稳定地位。

2003 年 5 月，在一次业内会议上，卡普尔与 AOL 执行官泰德·莱昂西斯（Ted Leonsis）不期而遇，晤谈之下达成了皆大欢喜的方案：AOL 从 Mozilla 中全身而退，但要资助一个新的非盈利基金会（每两年投入

200 万美金），该基金会负责支付 Mozilla 核心人员的工资并维持项目运转。卡普尔再另外投入 30 万美金，且同意担任基金会主席一职。

既然 Chandler 开发者们尚未取得长足进展，而且卡普尔又在 Mozilla 基金会身担要职，于是他建议再看看 Mozilla。他感到，这至少是个“大和合”的机会。



2003 年 7 月末的一个下午，文档架构组成员齐聚在沃德街 OSAF 办公室的一间小会议里。安德森打开自己最喜欢的个人信息管理器——一本绘有图画的纸质笔记簿。赫兹菲尔德的笔记本则形似账簿、系有一条红色书签带。卡普尔手边放一本标准拍纸簿。只有托伊用笔记本电脑。每个人都到新办公室的厨房里扯了一块比萨饼带过来。

急于推动 Chandler 项目的赫兹菲尔德出了个激进的点子：他指出，Mozilla 的架构已经可以容纳其他程序做插件。何不把 Chandler 直接做成 Mozilla 的一个大插件呢？当然他也承认这样做会有些问题。如果基于浏览器做设计，自然需要重新考虑 Chandler 的目标。不过，依靠浏览器界面的话，就可以省去程序员们自行搭建新界面的大量劳力。

“有那么多事要做。要是能绑上一支助推火箭就好了。”他说道。

迈克尔·托伊常常力主采用 Mozilla 部件，不过这回他提出警告。“互联网发明简直已有四万年，我们还是没办法让脑子不太灵光的人用好网站。而且网页浏览器对于非网页的东西来说也并非是很好的界面。就像是出征之前就绑了个劣质背包。”

安德森对比了 Mozilla 和 OSAF 已经开始采用的 wxWidgets 工具，他说，这种“苹果和橙子”的比较让人沮丧。“我偶尔会觉得自己是在冰山上漫游。”

卡普尔站起身来，走到白板前，画了一些方框，往上写标注。“这是屏幕。这些方框里面有些地方要放上丰富的小挂件，吸引用户注意力，也要与数据模型连接。表格。大纲。不同视图。这是我们偏好之物。我们知道自己要这些东西。该把这些东西摆上去。还有其他标准 GUI 部件——装饰性的文字、按钮和其他控件。所以，问题是，**这些东西**”——他指向挂件集的部分——“从哪儿来？原本我们打算用 **wxWidgets** 来做。问题在于，要花多少时间才能完成？”

安德森亢声道：“现在，做布局最简单的方式就是容器（sizer）了。”（容器[sizer]是 **wxWindows** 工具集用来在屏幕上放置窗口形状的基础单元。）

“事实证明这会比做网页难上一个数量级，”赫兹菲尔德反对说。

“我注意到容器是二维的，”托伊说，“而 **CSS**（层叠样式表，现代 **Web** 浏览器使用的布局规范格式）则能让你可以做置顶效果。”

他们讨论的时候，卡普尔正往白板上写：*容器框架足够灵活吗？足够有表现力吗？*

“使用 **Mozilla** 的部分原因，是便于自由表现布局上的一些元素，”卡普尔说。

“好些人觉得 **CSS** 太难，”安德森说。

“你需要一个对用户友好的编辑器，”卡普尔说，“没有好编辑器的情况是偶尔发生还是大抵如此？”

“迈克尔的意思是说，”安德森答道，“为 **CSS** 打造好的 **GUI** 基本上很难。”

“我们可能过于武断地决定不要解决富文档问题，”托伊说，“我能感到那是个黑洞。”

卡普尔继续写：*要解决富文档哪个部分的问题?*

每位与会者或多或少都感觉大家是在“摇手”——泛泛而谈，不强求细节。最热烈的“摇手”看来是围绕“文档”这个词的意义发生的。

“能把整个应用程序当成文档吗？”卡普尔问道。

“问题是，能否把它当成**文档集**？”赫兹菲尔德说。

“Chandler 是个文档吗？”托伊问。

赫兹菲尔德答道，“我们讨论的是对话文档吗？它们是不是要投射到屏幕上的基础文档的一部分？或者是隔离开的文档？”

“嗯，”安德森补充说，“还有镀铬区[①]（应用程序窗口周围的边框，放置菜单、滚动条等）。”

“自由意志存在吗？”卡普尔开玩笑说，“问题很多，不过既然有形而上的成分，我看不妨到无路可走再说吧。”[②]

四个人开始讨论基于 Mozilla 的 Chandler 如何提供类似“进度条”——一种逐渐填满的空条，在执行类似接收大尺寸邮件时实时告知用户进展程度——这样的简单特性。此类实时更新的动态界面元素，在一般程序中司空见惯，对于 Mozilla 等面向 Web 的平台则需要扩展，因为这类平台被设计成静态显示从服务器端取得的数据。基于 Mozilla 的 Chandler 得通过 JavaScript 实现这样的特性，JavaScript 是一种内嵌于网页的编程语言（发布网页的基础语言 HTML 并非真是一种编程语言——它只告诉浏览器在屏幕上显示些什么）。

“在 Web 应用中从没见过进度条，因为客户端与服务器之间每秒只能有几次数据往返，”卡普尔说，“我想确信咱们能有个好法子来做到这个。对我来说这是 PC 与 Web 的关键不同——每秒只能跟服务器有几次交互，否则程序性能就会崩溃。”

[①] chrome，这个区域容纳了一些界面元素，就好像是在外面镀了一层金属，表示“内有菜单、工具条”等。

[②] 形而上的自由意志是否存在，是哲学上的大问题。卡普尔用哲学术语开了个晦涩的玩笑。

比萨饼放凉了。讨论持续了一下午。卡普尔说，他打算请 Mozilla 的首席架构师布仁丹·艾克[Brendan Eich]（在网景公司工作时，他发明了 JavaScript）来和小组成员开个会。

“目前很难判断在 Mozilla 整合方面是否会有巨大进展。不过我想看看这次对话是否值得，”他总结说，“会是个大手术吗？同时进行心肺移植虽然极端倒也还有可能。如果要移植脑袋，就根本不值一谈。”

卡普尔告诉其同僚们，无论如何也不能止步于此。要在一个月内搞掂它。



一个月后，问题真的解决了。文档架构组放弃了在 Mozilla 之上构建 Chandler 的想法，决定回归原来的计划。“*文档*”一词仍然晦涩难明，当工作核心缩减到 Chandler 布局引擎的细节——不是“Chandler 的布局会是怎样？”而是“用什么工具做布局，程序员如何操作它们？”——时，他们用另外一个词定义自己的工作。他们正在为之努力的系统该叫做 CPIA（念起来像是“sepia[棕褐色]”），意为 Chandler Presentation and Interaction Architecture（Chandler 外观及互动架构）。

CPIA 将搭建于 wxWidgets 之上，但其目的却是把 Chandler 的 Python 代码包装到被称之为“构建块”的程序包中、连接到每个 wxWidgets 元素。CPIA 的构建块可以让非程序员轻易地制造 Chandler 扩展，只要把已有的构建块挂接到一起就可以了。就眼前而言，OSAF 程序员们也能用 CPIA 构建块来搭建 Chandler。程序将在 Chandler 资料库中保存构建块：这种“数据驱动”的设计理论上能更容易地修改构建块的特性——不用编写新代码，只需要修改和保存数据就可以了。

CPIA 是可复用软件部件乐高王国之梦的具体而微者。在 wiki 上修

订了数月之久的 CPIA 设计文档宣称，“构建块被设计为互相组合，就像乐高积木一样。”对于约翰·安德森来说，CPIA 背后的概念代表了关于更好的编程模型的长期愿景。他终于有机会借 Chandler 愿望成真。卡普尔也抱有同样的梦想；在我较早时对他的一次采访中，他描述过关于用类似乐高积木的部件组合成 Chandler 的想法，“顶底相接，从里到外”。

对于程序员来说，乐高积木的概念具有不可抵抗的吸引力。但 OSAF 团队并没有采用别人做好的乐高积木直接搭建；他们要从头做一套自己的积木，逐个做出构建块。

商业哲人彼得·德鲁克写过一个关于工作的寓言：

人们在管理会议上常说起一个故事，说有人问三个石匠他们干什么工作。第一个石匠答道：“我在谋生。”第二个一边干活一边说：“我在做全国最棒的石匠活。”第三个抬起头来，眼里充满幻想的光芒，说：“我在建造一座大教堂。”

第三个石匠自然就是真正的“经理”。第一个石匠知道要从工作中得到什么，而且努力去做。他是那种做一天和尚就好好撞一天钟的人。

第二个石匠是个问题。手艺是基本要素……不过即便是真工匠和真专业人士，也会在磨光石料和收集脚注时错以为自己在做什么了不得的事。

卡普尔把 CPIA 叫做“基础架构投资”。但它疑似磨斧头或剪牦牛毛。安德森及其同事只是在“磨石头”吗？这可无法预料。



8 月的一个下午，安迪·赫兹菲尔德和约翰·安德森坐在沃德街办

公室的一间小会议室里。他们谈到赫兹菲尔德关于代理的点子。他们提到“相关场景”，一种围绕用户活动相关类别组的 Chandler 数据组织新概念。他们谈到安德森编写灵活的界面构建包的进展。然后他们面面相觑。

“事情一有进展我就会兴奋，”赫兹菲尔德说，“我自己的风格是动手太快。不过也许这能拉动整个团队。”

“我尽量谋定而后动，”安德森答道。

赫兹菲尔德笑了。“比尔·艾金森总说我老是‘准备、开火、瞄准！’”

“以前我都是自己做完整个项目，”安德森说，“我不善于拆分工作，让许多人协同工作。”

赫兹菲尔德问，在 NeXT 公司也这样吗？

“我在那儿当经理时，自己不写代码。这和你只是五个人中的一个、写五分之一代码可太不一样了。我要么自己写大部分代码，要么管理团队。”

安德森沉默片刻。通风系统嗡嗡作响。“我有点完美主义者倾向。”他合上笔记簿，站起身来。“现在我想要干的，就是把这个做起来。”



在软件世界中，*集成（integration）*的意思就是把一段运行正常的代码连接到某个程序中另一段运行正常的代码上。集成往往是软件项目遇到大麻烦的环节。分开来运行相当正常的代码，在合起来时就闹罢工：不能正确挂接、发送不可解释的消息、或者顽固地拒绝启动或停止。（这是程序员的造物模仿其人类造物主的形式之一。）

Chandler 现在正往这种集成点蹒跚前进。当 CPIA 的发动机在后台

轰鸣时，迈克尔·托伊也开始推动 Chandler 开发者们沿着他在当年夏季早些时候划定的轨道前行，目标是在 9 月底发布“时钟驱动的”Chandler 0.2 版本。OSAF 已经习惯于每周或每两周为 Chandler 设定一个新里程碑，但发布一个“小数点后一位版本”却兹事体大。所以，随着 0.2 版发布日期日益逼近，开发者开始尽力从自己的工作——如安蒂·维达的新资料库，或者安迪·赫兹菲尔德的代理——中获取成果，并将其加到 Chandler 的“主干代码”里。

现今的多数项目都接纳了“持续集成”的观念：程序员不断向主干代码中签入最新的代码，每个人负责确保自己加进去的代码不会导致问题发生。在稍后一段时间，OSAF 也将采用一种更高层面的持续集成，但 Chandler 0.2 版则更像是软件开发分析师们所说的“宇宙大爆炸式集成（big-bang integration）[①]”：所有人都尽量在最后集成代码，结果总是一塌糊涂。

托伊清楚危险所在，但他感到他们应该集体承诺每三个月发布一次 Chandler 版本，而且应该按时做到。尽管维达的新资料库已具备集成条件，但将它全部放到 Chandler 0.2 版本中时间还是略显仓促。只有赫兹菲尔德的代理程序部分的支持性技术框架已经准备好。CPIA 则还只停留在蓝图阶段。

然而期限就是期限，所以开发组只能尽其所能准备 Chandler 的新公众版本。在“代码冰冻日”之后，他们不再签入任何与新特性有关的代码，而是集中精力修正明显的缺陷并为新版本编写一些文档（给其他开发者和用户阅读的解释性资料）。

Chandler 0.2 于 2003 年 9 月 25 日发布，距 OSAF 首次公开 Chandler 计划已将近一年。它的发布伴随着质疑。下载和试用了这个版本的少量用户惊奇地发现，它的功能比 Chandler 0.1 还要少。它就像是在一个垮塌后又部分重建的架子上遮了个盖子。

[①]按照现代宇宙学中的大爆炸理论，宇宙是从温度和密度都极高的状态中由一次“大爆炸”产生的。时间至少在 100 亿年前。

开发者们，尤其是维达，深感沮丧。“Chandler 0.2 即将推出，不过我恳请你们别下载，”迈克尔·托伊在 9 月 19 日的 blog 中写道，“我是说真的。别下载，别告诉朋友，别在 Slashdot 帖子中加下载链接……”托伊解释了为什么要正式发布这么一个未完成的版本：“在开发周期的前期，只要有一次像个真正的应用程序那样运行起来，我就该相信自己的数据，马上就开始操心软件发布带来的细枝末节似乎不算正解。不过，如果没有中间版本期限的话，就会导致在充满各种编码可能性的土地上漫无目的地四处游荡。”

在发布前一周的一次开发者会议上，“鸭子”舍伍德问道，“我们要开个 0.2 版庆祝派对吗？”

“也许开个守灵会吧，”托伊回答说。

“不如默哀片刻，”卡普尔建议说。

10 月 2 日，开发者们开会检讨 0.2 版。出了什么问题？下次应该怎么办？

摩根·萨奇建议说，0.2 版真的只应该当作是一个临时版本。“小数点后一位版本应该包含新特性的。”

“当秒表掐停时，我们半在人间、半在地狱，”安德森说。

布莱恩·史金勒嗓门大起来。“有些特性没加进去，但也另外加入了成吨的特性。而且我们现在有了资料库，以后还要靠它工作呢。”

开发者们一个接一个发表批评意见。当 Chandler 的架构仍摇摆不定时，停下来修正缺陷是否值得？Chandler 大部分功能还无法工作时，怎么能就想到“安装完毕”？是否能在 15 个月之后的 2004 年 12 月准时推出代号为 Canoga 的 Chandler 1.0 版本？

托伊在白板上写下这些批评意见，每条前面标记上一个箭头符号。

“我们都同意，版本发布必须有目标，”托伊说，“但 0.2 版的确没有目标可言——它造成了影响，但却什么也没做到。雷声大雨点小。没什么可庆祝的，也就不该开什么派对。”

OSAF 不该再有 0.2 版这样令人沮丧的版本，这是因为已从中吸取教训，还因为 OSAF 将永远不再单纯根据发布期限来做版本计划。以后每个版本都要围绕实现一系列特性的目标展开，进度计划应按需制定。0.2 版的失败体验没有让 Chandler 的工作偏离正轨，但确实给 OSAF 留下了一道伤痕。

卡普尔从 0.2 版中吸取的一条教训——该死的布鲁克斯法则——是如果 OSAF 想要在可接受的时限内发布 1.0 版本的话，就得招募更多程序员。在 0.2 版检讨会上，已经有几个新面孔出现，而更多人马很快就会到位。沉默寡言的芬兰人赫基·托伊沃伦（Heikki Toivonen）是 Mozilla 项目的长期贡献者，后来成为网景雇员，现在负责实现 Chandler 的安全特性。史都沃·帕门特（Stuart Parmenter）在网景公司被誉为常驻神童；读高中时，在网景公司决定开放浏览器源代码后不久，他就已经开始成为 Mozilla 的志愿贡献者；18 岁时，他从乔治亚州搬到硅谷，为网景公司工作，直至 AOL 浏览器部门关门大吉。到 OSAF 后，他和赫兹菲尔德一起开发代理功能。

另一位新人是个叫做尹咪咪（Mimi Yin）的年轻女士，她安静地坐着，倾听会议上的集体自责。她任 UI（User Interface，用户界面）设计师一职。在耶鲁大学念音乐和舞蹈时，她就为学生健康服务中心有偿设计网站和群发电子邮件。1999 年毕业时，正值互联网泡沫高峰期，她移居到旧金山湾区，为一家电子商务公司工作，同时学习界面和信息设计。互联网泡沫破灭后，她一边在纽约做舞蹈编排，一边在西海岸从事设计工作，最近的一份工作是承包微软 MSN 在线服务的重新设计项目。



尹咪咪

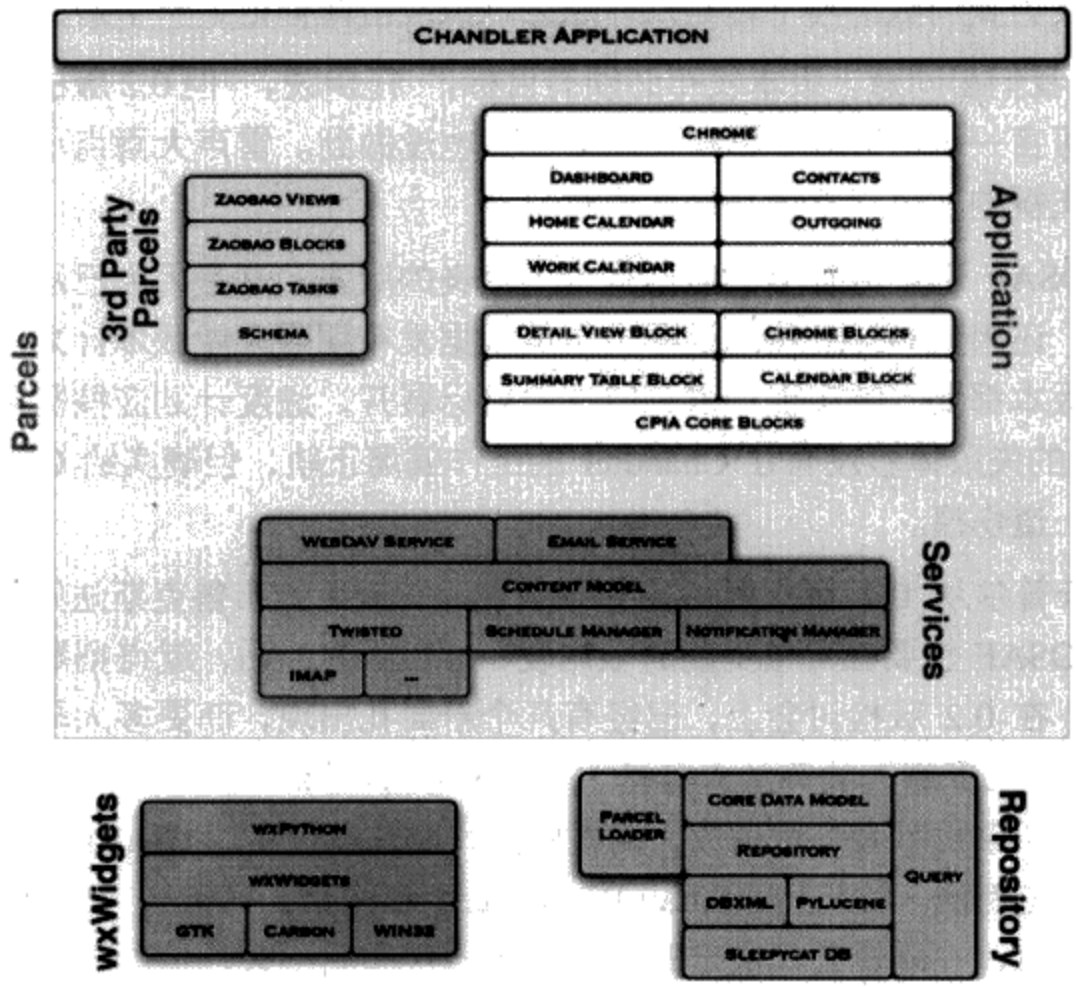


图 6-1 Chandler 的架构

2003 年 5 月，她曾到 OSAF 面试过，不过失败了。她精明而博学，但有些面试官觉得她还欠点经验。到了秋天，林智超打电话给她，邀请她加入 OSAF。“我猜他们还没找到那个有魔力的人，”她想。



尹的到来催动设计工作重新进入高效轨道，这正好赶上 2003 年 10 月开始的 Chandler 0.3 版开发周期。新的设计组成立了：成员是卡普尔、赫兹菲尔德、林智超、“鸭子”舍伍德、布莱恩·史金勒和尹。他们开始

每周花三天下午开会，讨论 Chandler 的一些核心理念——例如消灭数据地窖。尹擅长在白板或 wiki 上快速绘制出头脑风暴成果的纲要，这有助于总结讨论、凸显隐藏的问题。

OSAF 灵活的工作时间表允许尹早晨上舞蹈课，下午再赶来参加会议——并在她方便的时候往 wiki 上写东西。她开始在公众场合自言自语、往 wiki 的 Jungle 部分添加大量页面，用骆驼标记法风格的文字公布头脑风暴成果：

BagsOfAttributes
PIMParadigmShift102003 HowCloselyTheUIShouldMapToTheDataModel
BalancingPointFoundWhat DoesItMeanToGetTheBasicsReallyRight
WhatKindOfMessHaveWeGottenOurselvesInto

开始在 OSAF 工作后不久，尹参加了戴维·艾伦（David Allen）主讲的一个全天课程。戴维·艾伦是一位生产力教练，其著作《搞掂（Getting Things Done）》一书[①]在程序员人群中近乎圣典。卡普尔认识艾伦——他们都对个人管理方案感兴趣——而艾伦也到过 OSAF，向大家展示其个人管理原则（拥趸们称之为 GTD）的电子版本。但尹却是 OSAF 团队中愿意系统考虑 GTD 观念能如何帮助打造 Chandler 的人。

《搞掂》一书写道，现代生活和工作让我们在精神上受困于大量未完成的任务——周围要处理的事物、要答复或过滤的电子邮件、应该阅读的文档或出版物、答应别人要做的事。这一大堆“开放循环”加起来就是艾伦称之为“填料”的东西。GTD 认为，通过创建一套“信任系统”——写在纸上或电子化都行——可以让人不再被这些填料压垮，成为外部事物的主人。

把填料塞到这套系统里面，就不用自己去操心了。在系统中为每个项目定义“下一步操作”。使用这套系统，最终能达到武术迷艾伦所谓“心

[①] 全名 Getting Things Done, The Art of Stress-Free Productivity

如止水”的境界——从焦虑导致的过压状态和“该死，我还忘了做什么？”的自寻烦恼中解脱出来。

GTD 的核心原则之一建议有规律地安排定量时间处理收件箱：查看胡乱塞进系统的每个比特填料并做出决定，“下一步怎么处理这东西？”艾伦建议，如果在两分钟之内就能处理好，马上就开始做。否则，决定是存档、丢弃、推迟，或者分类到某个有下一步操作建议的具体项目中。

有了卡普尔的认同和鼓励，尹咪咪开始考虑 Chandler 怎么能像 GTD 系统处理填料那样工作。卡普尔明白自己并不想把 Chandler 太过绑定在 GTD 上，但当设计组进入 2003 年秋天紧张的工作季时，总是周期性地回到艾伦的理论中寻求灵感。尹想出了一种“仪表板（dashboard）”界面概念，它能帮助 Chandler 用户应付蜂拥而至的电子邮件和其他条目，而不必马上决定如何处理它们。尹的仪表板是一种全新的收件箱界面，它以“过去”、“现在”和“未来”的组织形式向用户展示一个自顶向下的条目列表。旧条目将从列表上端滚动出屏幕并进入存储区；延迟或放在未来处理的条目从屏幕下端消失；而需要处理的条目则停留在屏幕中央以引起用户的关注。尹的仪表板创意来自于面向时间线的界面设计传统，这种设计观念主张以时间顺序为信息管理的组织原则，如戴维·格林特（David Gelernter）的 Lifestreams 项目那样。但它又融合了 Chandler 的无地窖概念和戴维·艾伦的 GTD 原则，最终炼成一种独特的东西。

此时 OSAF 自身都还搞不太掂，但即便有人注意到这种讽刺性现象，也没把它写到 blog 或是 wiki 上去。尹开始做 Chandler 用户界面设计的基础工作，《搞掂》书中写到的一种现象或许正可得到印证：“事情很少因为时间不够而停滞，而是因为没有确定怎么做而停滞。”尹的到来意味着 OSAF 终于有了一位专职人员，负责就 Chandler 的用户界面提出问题并做出回答，以及定义这个界面。一方面，这意味着——尽管设计组还

没和程序员们分享成果——软件开发者们终于感到很快就会拿到一份完整的设计方案了。另一方面，随着尹了解并跟上了 Chandler 项目之前的工作（布鲁克斯法则在设计领域又起作用了），开始贡献新的好点子，设计组的成员们发现自己回到了第一原则上。

Chandler 不断地重复回归。在较早时的头脑风暴会上，卡普尔和摩根·萨奇就是否采用 RDF 的问题苦苦纠缠。还有那些关于设计一套能在网络上正常工作的灵活资料库的冗长争论。还有文档架构组探索的扩展领域。现在，OSAF 团队再次提出了一个基础问题：人们如何组织信息？一套创新软件工具如何能帮到他们？



图 6-2　尹咪咪借鉴 GTD 做的仪表板设计

这对安迪·赫兹菲尔德来说难以忍受，他再也无法忍耐 Chandler 项目的乏力状况。关于项目，他感觉不好已经有一段时间。他后来告诉我，首先是他发现，作为一个志愿者，很难和专职雇员一起工作——即便是为卡普尔那样自己乐于相与的人工作也是如此："他们不知道该不该指望你。而且，你不拿工资，也就不受控制。"他还觉得尹的方案过于抽象、过于学院派，一无所用。

但真正的问题是速度。"我是那种喜欢一路扔避雷针的开发者。要做大程序，至少得有一个人与之同呼吸共命运。全情投入。但事实正是少了这样一个人。" 2003 年早些时候，因为受不了 Chandler 过慢的开发进度，他曾为 Chandler 做了一个基础的地址簿功能包："我自己设计了一下就写出来了。也没跟谁讨论。我的做法就是，写代码，尽力而为，一切都有可能随着学到的东西而改变。但你是在从难题中学到东西的阶梯上攀登。每个难题都有其独特之处，所以要逐个学习，再做事情，从而占据有利位置。站在有利位置，就能决定抛弃什么——代码不值钱。但它常告诉你下一步该怎么做。"

不过地址簿从未集成到 Chandler 代码的主干代码中。实际上，很久前卡普尔就已决定不再强调 Chandler 的地址簿功能；或许将来可以有这个功能，但现在其优先级却很低。所以当赫兹菲尔德加入到那年夏天的文档架构组马拉松会议时，就决定全神贯注于充实 Chandler 代理的点子。

秋天，他和妻子一起到夏威夷度了一个月的长假。回来时，设计组的讨论结果让他大吃一惊。他觉得他们根本就忽略了文档架构会议的所有成果——辛苦得来的词汇表、仔细做出的构建块和屏幕布局的草图——并且重起炉灶。而这似乎就是卡普尔想要的，真令人费解。

“我跟米奇谈起时，他说：‘嗯，我想试试不同的东西——我们要转头回来重做规整。’且慢！他们已经做出来了！所以我假定米奇放弃了CPIA。其实我搞错了，他并没放弃。不过当时我没这么觉得。我想，尽管米奇鼓吹CPIA，但他对CPIA及其制造过程还是有点不满意，所以他想要来点新想法。不过，这让我觉得事情总在绕圈子。我对米奇仍抱有无限的崇敬。可他却弃我不顾。”

2003年最后那段时间，赫兹菲尔德从OSAF抽出一些精力，自己开始一个新项目：在名为Folklore.org的网站上构建一套源自blog和wiki的小软件，让同好者可以贡献和共享故事。然后他开始在网站上撰写关于自己在苹果公司时如何帮助创建第一个麦金塔操作系统的事情。故事越来越多，站点发展起来了。计算机图书出版商蒂姆·奥莱理（Tim O’Reilly）——其公司专攻开源开发文档，另外，他也是OSAF的董事会成员——看到Folklore.org，建议赫兹菲尔德把这些故事写成书。

赫兹菲尔德同意了。他得在6个月内完成原稿。除了给些建议和偶作拜访，他在OSAF的工作已告完结。

10101010101010101101010110

自2003年夏天开始，不断有消息从微软位于华盛顿州雷德蒙得的总部传出，让世界得以窥见代号为Longhorn的Windows新版本软件。计算机行业“远程预警系统”媒体的读者们了解到，Longhorn将绝对令人难以置信。它会拥有一套十分漂亮和快速的新图形系统，叫做Avalon。它将提供威力十足、神通广大的新网络系统，叫做Indigo。而且在核心部分它还会有一个扩展性和创新性惊人的新文件系统，叫做WinFS。

在采访中，比尔·盖茨鼓吹Longhorn的能力。他所说的功能与卡普尔为Chandler设定的功能相当类似，包括消灭那些隔离不同类信息的

“地窖”在内。WinFS 文件系统甚至将允许你将同一文件存储在几个不同地方！

“在机器间转移文件、共享电子邮件和日程表——这真令人痛苦，”盖茨对《今日美国（USA Today）》记者说，“比如，你有个工作日程表，还有个家庭日程表。是否能真正轻易地调整家庭日程表，看看哪些事件应该在两张日程表上呢？Longhorn 让你的信息能方便地在不同设备上显示。它让你方便地浏览信息……如果问某人今天你怎么用 PC 管理股票、电影、音乐、餐馆信息，他当然可以做到，只是十分痛苦，因为手工工作太多。系统天生不明白你生活中要应付的这些信息。你到计算机上找指导，找到这个有趣的网页，也许打印出来。这种为离线时阅读而保存的想法——无论如何，好多这类事仍然很麻烦。所以 Longhorn 会改变用户界面，把许多分离的东西统一起来。”

微软还没笨到说定 Longhorn 发布的具体日期。该公司 Windows 升级版本的跳票记录已是声名狼藉。但 Longhorn 的鼓声对于卡普尔和 OSAF 的所有人都是一次警醒，交付 Chandler 所花的时间越长，用这个项目改变世界的机会就越小。

自卡普尔建议说 Chandler 1.0 应当在 2003 年底完成已有一年之久，现在看来到 2004 年底发布都有点难。他开始和林智超协作，对 Chandler 冗长的特性列表进行优先级排列。Chandler 之前的功能涉及许多广泛的领域——电子邮件、日程表、任务、联系人、共享、同步、搜索、安全、代理——而且每个领域又有长长的子特性列表。林智超把所有特性整理到一系列表格中，并在稍后移到 wiki 上，他和卡普尔开始决定先做什么、后做什么。

卡普尔越来越关注招聘事宜。夏末，他在 blog 上贴文说：“我们几个月来一直保持在 12 人左右……我们的计划是最多招个 30 来人，好在接下来的两年半里完成 Canoga 和 Westwood 版本。”但寻找合适的开发

人员并不容易，将其整合到 OSAF 团队中也需要时间。“在不引起我能接受的更多混乱和破坏的前提下，每个月我们平均只能吸收两个、偶尔是三个新人。”他写道。

10 月，迈克尔·托伊向卡普尔提交辞呈，这让卡普尔颇感意外。迈克尔·托伊有几个朋友开了家新软件公司——改进呼叫中心业务、让工人在家接电话、并把所有流程迁移到网上。他们请他做副总裁。虽然他把 OSAF 看作梦幻工作，但他还是惊奇地发现自己被这个职位所吸引。

尽管托伊的自谦常被看作是一种叫人喜爱的性格——在一张公开的周工作列表上，他写道，“继续集中于努力与团队成员沟通，依其所述，作为经理我还是能帮到他们的”——但他还是诚实地感觉到自己没干好。后来他向我解释了离开的决定。

“米奇·卡普尔每年在这上头花了几百万美金，而且他不想扑到细节中去——他想找个人来保证钱没白花。而那却是我在细节管理和产出保证上都无法做到的。米奇真的是很耐心，总是说我会帮助你搞好这些事。所以我也对自己的职位毫无怨言。但在最后，我被要求提供我不擅长的服务。”

“我坦承自己强烈倾向于尽快交付产品并从中得到经验教训。这会更好玩、更有趣，而且，我想这也会是比确知工作会持续十年、十五年之久来得好的做事方式。看起来 Chandler 要尽力保证架构上可靠，而且为了有个好架构宁肯无限推迟面世。那是米奇的钱，他愿意做这个交换就行。放慢速度可能会在未来获得丰厚的回报。但对于想在下周就发布软件的人来说实在是难以忍受。”



托伊的离开按部就班、和和气气——11 月份的多数时间他留在

OSAF 收拾首尾——不过还是让卡普尔陷于困境。上次用了几个月来挑选合适的开发经理。这回卡普尔倒是碰巧马上就找到一位强有力的候选者；他面试顺利并接受了工作邀请——但过了几天却打电话来说，对不起，他老婆不愿意搬到湾区（Bay Area）。

寻找经理将是旷日持久之事，OSAF 正开始投入 Chandler 的设计当中，并越来越现实地看待其优先级，这些都让卡普尔烦恼。他宣布说，还要继续找人，但不能因此而影响进度。显然，如果 OSAF 还坚持原来的每三个月发布一次新版本的时钟驱动进度计划、尽力在 2003 年十二月底发布 Chandler 0.3 版，则 0.3 版决不会比 0.2 版更让人满意。但开发者们正在努力将安德森的新 CPIA 构建块放置到位，设计会议卓有成效，卡普尔决定更重要的是做一个有显著进步的 0.3 版，而不只为掐发布时间。

在托伊效力于 OSAF 的最后一个星期，凯蒂·帕兰特得到授权，担任 Chandler 的软件架构师，即约翰·安德森原来的职位。托伊走后，卡普尔决定把 OSAF 的开发工作划分为三个小组：设计组；资料库组；应用组，在这个部分会有一大群 OSAF 程序员打造 Chandler 的前端。帕兰特在应用组正式成立前就暗地开始做组织工作，她将负责这个核心小组。

卡普尔宣布，在找人替代托伊的工作的同时，他亲自负责整体开发管理。

“我将前所未有地关注开发细节，”他告诉团队成员们。“希望能有点用，别帮倒忙就好了。”



行文至此，我知道，每位读到这本书的开发者都会绝望地将它扔到墙角，想，“别再继续发疯了！在书里，他们该错的都错了！”

我猜，许多程序员还会想，“我没这样干过。我做得比他们好。”

他们中的许多人可能都说对了。

在近距离观察 OSAF 工作一年之后，我发现自己也在怀疑，“他们何时才能起步？要花多长时间？障碍是什么？”软件时间的熵牵制效应起作用了，看起来没什么能加速进展。

难道 Chandler 团队看不到自己已经偏离正轨有多远吗？在项目的每个转折点，卡普尔总是坦言，如果他一早就知道后来学到的东西，进展就会有所不同。他耸耸肩，叹道，“马后炮总是打得准。”跟着就继续工作了。

在软件编年史上，Chandler 缓慢的步调并非例外，反倒是意料中事。在这个领域，记录显示每位司机都会选择不同的路途偏离正轨，但几乎所有人迟早都要栽到沟里。

假如您就是一位确信自己可以做得更好的程序员读者，我想请问，在上一个项目中，您有多少次扪心自问，“嗯，我们原本或许**应当**这么做”——而“这么做”则是软件领域某种最佳实践标准——“不过这是特例，我们的情况有所不同。”就像安迪·赫兹菲尔德写的那样：“没有所谓典型的软件项目。每个项目都不同。”

2004 年 6 月，《Linux 时报（Linux Times）》发表了一篇对 Linux“仁君”李纳斯·托瓦茨的采访。

“你对那些刚开始做大型开源项目的人有何建议？”记者问道。

“别做大项目，”托瓦茨要言作答。

“从小项目开始，而且永远不要期望它变大。如果这么想，就会做过度设计，把它想象得过于重要。更坏的情况是，你可能会被自己想象中的艰难工作所吓倒。所以要从小处起步，着力考虑细节。别去想大图景和好设计。如果项目没解决某些眼前的需求，多半就是被过度设计了。”



李纳斯·托瓦茨

托瓦茨没有点 Chandler 的名，但熟悉该项目的人都难免会联想到它，而那些做 Chandler 的人耳闻此言则多半会打个寒颤。

即便你采纳了托瓦茨的建议，即便从小处起步、压抑自己的野心、考虑细节并永不幻想宏大图景——即便是那样，托瓦茨说，你也不该指望进展迅速。

“别指望在短时间内达到大成就，”他宣称，“我致力于 Linux 达 13 年之久，我想后面还得花上好些时间。如果一早就**妄想**做个大东西，可能现在还没动手呢。”

第 7 章 细节视图

[2004 年 1 月～5 月]

豆大的雨点打在沃德街大会议室的窗户上。窗帘拉下了；投影仪在嗡嗡作响。这是 OSAF 有史以来第一个“演示日”，时间是在 2004 年 1 月一个阴郁的早晨。杰德·伯吉斯的计算机屏幕投射到屋子后墙，所有的眼睛都盯着。

约翰·安德森开始说话时，伯吉斯启动了 Chandler。启动花了令人难以忍受的 60 秒钟。安德森扯东扯西、消磨着等待的时间：“大家会留意到，启动不太顺畅。”最后，窗口终于显示出来了。

“这就是新的资料库查看器，”安德森说，此时屏幕上满是一行行的 XML 数据。“它完全基于我们的新架构。”

在外行看来，这袒露底层的图景远远不够深入人心，但屋里的看客们却都全神贯注。他们明白，Chandler 中的这小小的工作组件——从资料库中获取数据并以原始形式显示给用户——是用 OSAF 自身的部件做成的，在过去几个月里，他们竭尽全力设计和打造了这些构建块。

“CPIA 构建块最酷的特性之一是它几乎全是数据，只有少量代码，”安德森说，“这与过去开发用户界面的方式背道而驰。我们走向了一种新模式，让 UI 元素基本上变成可以检查的数据。”换言之，用户在 Chandler 屏幕上看到的所有东西——每个窗口、按钮和菜单——都代表了 Chandler 资料库中的一块数据。

“那你能让这些修改即时生效吗？”米奇·卡普尔问。

“可以——而且不用写代码，”安德森答道——只要改一下数据，屏幕上就会发生变化。

“即便在这个层面上，也算是一种胜利，”卡普尔说。

演示日看来有点效果。卡普尔想起 0.2 版发布时陡然降临在团队的种种不爽，他决定在每周例会上留点时间，让程序员们互相展示成果，回顾最近达成的每个小进步。与会的两打人向安德森和伯吉斯鼓掌欢呼，一种轻快的感觉也在屋里升腾起来。

接下来，摩根·萨奇描述了他新近做的 Chandler 自动测试系统。他桌上摆了三台计算机，分别用于测试 Windows、Mac 和 Linux 版本的程序。运行在这些计算机上的脚本持续查找新签入的 Chandler 代码；每次有开发者向源代码库签入了改动了或新增的代码，这些计算机就会进行测试，确保新代码没有“破坏构建”——确认新代码与原有 Chandler 代码组合起来没导致结果程序包不能通过萨奇的测试。这倒不能保证对代码的修改没有缺陷——远远不能——不过却是对抗程序毁坏和编程错误的第一道防线。

这三台“哨兵”计算机将测试结果发送到一个叫做 Tinderbox 的程序，Tinderbox 再把结果发布到网页上，用三幅图表显示出构建的当前状态。绿色表示通过测试；黄色表示有一些问题；红色表示程序崩溃了。开发者和好奇的旁观者都能看到这个页面。（你也可以在这个地址查看项目的当

前构建状态：http://builds.osafoundation.org/tinderbox/Chandler/status.html。）

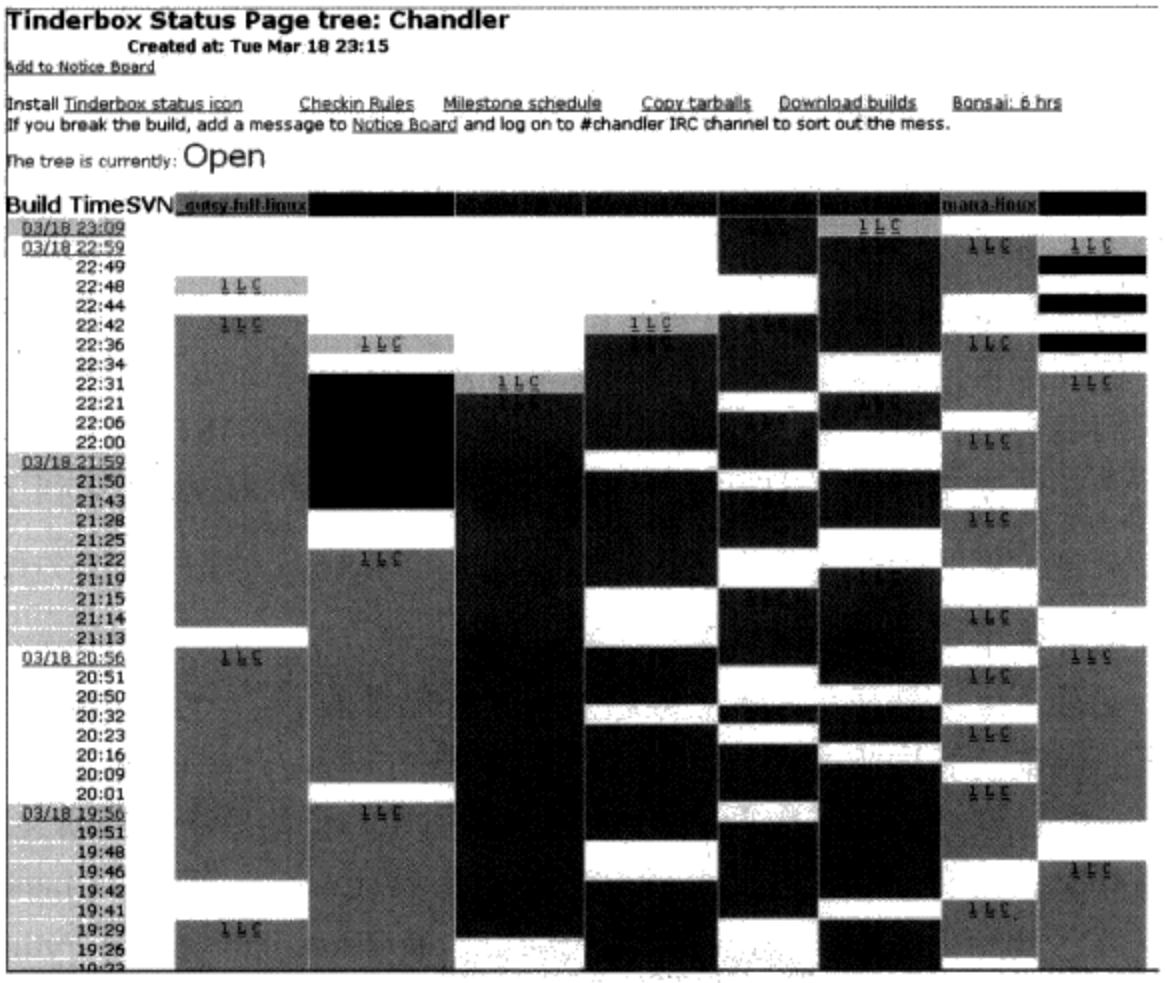


图 7-1　Tinderbox 显示了当前构建状态

下一个演示者是泰德·梁（Ted Leung），OSAF 最新的成员。梁曾在苹果公司为牛顿（Newton）手持设备编写软件；后来在 IBM，他参加了制订 XML 数据标准的早期工作。在互联网泡沫爆发期，他成为 Apache 开源项目的核心参与者。Chandler 的首次公告吸引了他，他开始关注这个项目，用“萌芽（sprout）”做网名、在 OSAF 的 IRC 频道上现身。他住在西雅图城外普杰特（Puget）海河上的孤岛贝恩布里杰（Bainbridge）；如果 OSAF 采取“面对面”工作方式、要求雇员在同一物理空间会面和工作，他就不可能到 OSAF 工作。但卡普尔明白，“真正的”开源项目



泰德·梁

应吸引散布于各地的程序员做出贡献，程序员之间通过互联网协同工作。他决定，是时候改变 OSAF 的工作方式了。梁在为 Apache 工作时学到了如何实施“分布式开发”。卡普尔邀请他加入 OSAF，在家工作，隔几个月到旧金山总部来一次即可。

今天梁要演示一些测试，检查 Chandler 资料库处理大量数据的能力。梁热衷于 blog 写作，他写了个程序，把多达 20Mb 的 blog 文章用 RSS 格式输入 Chandler，创建出 11000 个 Chandler 条目。只有一个问题：OSAF 的一位同事替他做演示，梁打电话进来参加，但电话工作不正常——他能听到旧金山，但旧金山听不到他。所以他只能通过在 IRC 聊天室中输入文字来做演示解说，说明文字投射到墙上，这样同事们就能看到了。

卡普尔刚开始时还为电话故障而丧气，跟着就为大家的创造力所触动。他慢慢地摇晃着脑袋说道：“这绝对是个创举。”

安蒂·维达是演示名单上最后一位。他先多演示了一会儿梁的资料库压力测试，然后切换到全文索引。他走到白板旁，开始讲解。如果要让用户能检索 Chandler 数据，程序就需要索引用户资料库中的所有文本。适逢其时，已有一种绝佳的开源文本检索工具 Lucene 存在，Chandler 可以用它。但有个阻碍：Lucene 用 Java 写成，而 Chandler 则基于 Python。怎么整合呢？

“我得演上几套杂技，”维达说，他脸上浮现出一种自豪的微笑，继续阐述——与会的非程序员被远远抛在身后。使用开源编译器 gcj 和一个叫做 SWIG（即“简化封装和界面生成器，Simplified Wrapper and Interface Generator”）的工具，经过一大串编译和“封装”，Lucene 终于能顺畅地与 Python 和 Chandler 资料库共同工作。跟着维达演示了用这套代码处理长达 2.5Mb 的文本文件——一份 1992 年的中情局世界报告。程序只用了 1.3 秒来索引多达 300 000 个单词——在这一点上比 Chandler 要快，足敷一切之用。

维达带领同事们游历他编写的代码巴洛克变换[①]时，眼睛闪亮、眼皮不停眨动。他说平安夜凌晨自己还在加班完成工作，听起来不像是抱怨；显然他乐在其中。

演示末了，卡普尔站起身来，热情高涨。“咱们在基础构造上投的赌注开始有回报了。”他宣布不再继续招聘开发经理。“我是个控制狂，还没准备好放弃这份工作呢。”他开玩笑说。不过迈克尔·托伊离去后新搭的组织结构显然管用。项目开始有了进展。代码也逐渐写出来了。在开发团队中，到处都能感受到一种弗里德里希·布鲁克斯称为“推挤（hustle）”的紧迫感。

[①]欧洲文艺复兴之后出现了巴洛克风格。它的特点是在文艺复兴建筑艺术的基础上强调新奇、变换和流动态的审美。



2004 年 2 月 26 日，OSAF 发布了 Chandler 0.3 版。新版本包括了 CPIA 的工作版和更成熟的资料库，但基本上还是没法用——只满足了可能对卡普尔及其团队进度好奇的外部软件开发者们的兴趣而已。但 OSAF 内部的氛围却与发布 0.2 版时全然不同：有 T 恤衫签名活动，在办公室的厨房区举行了一次小小的巧克力蛋糕庆祝会，有些 OSAF 的老成员，例如罗·蒙特利和阿历克斯·托蒂克都参加了庆祝。程序员们觉得自己有了动力。

这也意味着原来为达此目的而放任设计组自行工作的应用组开发者们开始渴求一张更彻底、更接近最终需要的 Chandler 外观和行为的路线图。随着构建程序真实用户界面——不是原型或临时版本，而是使用 Chandler 团队辛劳一年发明出来的真实资料库和架构——进入众人视野，程序员们对于信息的需求显而易见。

他们需要细节。他们需要蓝图。他们需要规格说明（specifications）——这个词对编程至关重要，随着时间推移，只留下了第一个音节，变成简

洁而不可再分的缩略词“specs（规格）”。

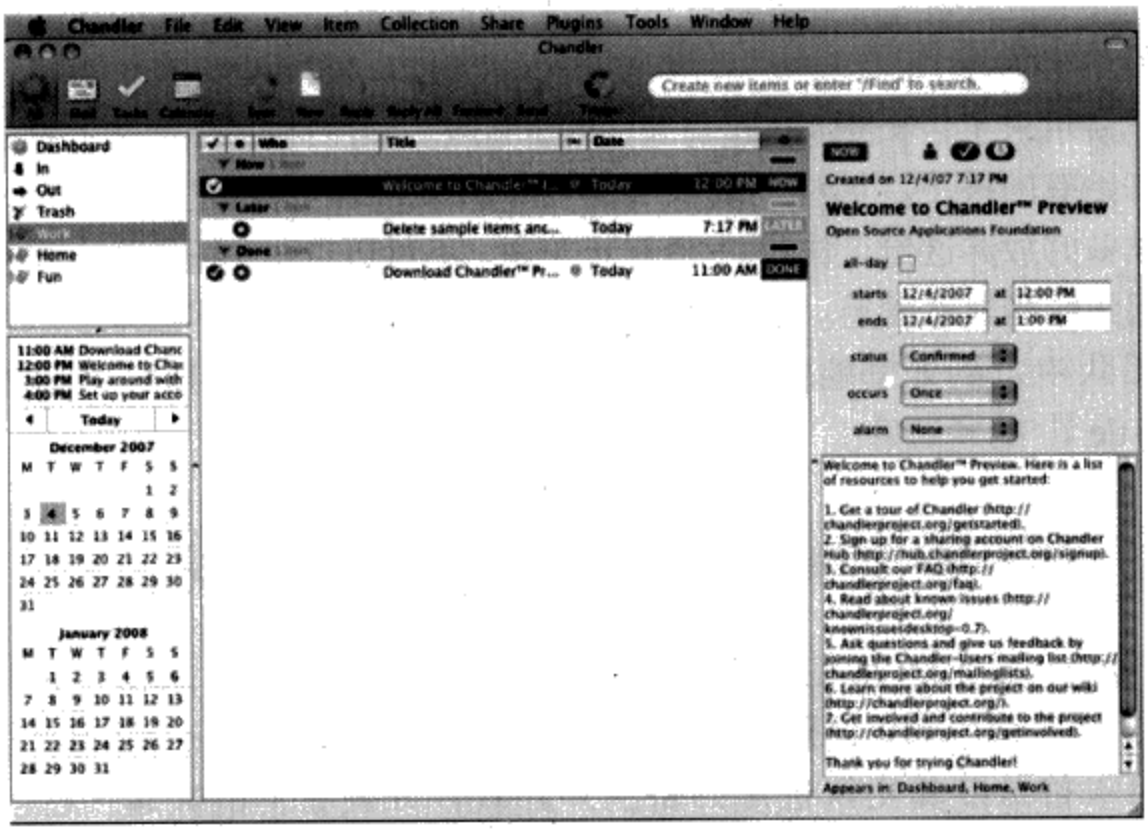


图 7-2 Chandler 0.3 版



如果你曾看过那部艳光四射的重金属恶搞片《摇滚万万岁（This is Spinal Tap）》[①]，也许会想得起那个彻底完蛋的乐队指望用凯尔特风格的巨石柱装饰舞台而重获新生的场景。当主要舞美装置从天花板降落到一群跳舞的侏儒中间时，演出停止了：巨石柱拱门只有一英尺半高。

“我认为问题不在于乐队不行，”演出失败后，主唱在旅馆里向经纪人抱怨说，“我想问题是舞台上的巨石柱有**被侏儒踏平**的危险！”

问题的根源？糟糕的规格说明。吉他手奈吉尔·塔夫内尔（Nigel

[①] 1984 年美国人罗德·瑞纳（Rob Reiner）自导自演的讽刺片。

Tufnel）深夜在咖啡馆里用餐巾纸背面草草描出设计图，一个多余的撇号就让十八英尺的巨石变成了十八英寸的石桩子。

“我的工作是完成乐队创意人员交代的工作，”经纪人抗议说，“奈吉尔给我的草图上分明写着 18 英寸。”

“但你没他那么糊涂，对吧，”歌手答道，“你做事不能像奈吉尔那么糊涂。”

当经纪人责骂布景师时，她出离愤怒了。她从公文包里抽出那张餐巾纸：“看吧，看吧，这就是你们要我做的。18 英寸。就在这儿。**写明了的。**”

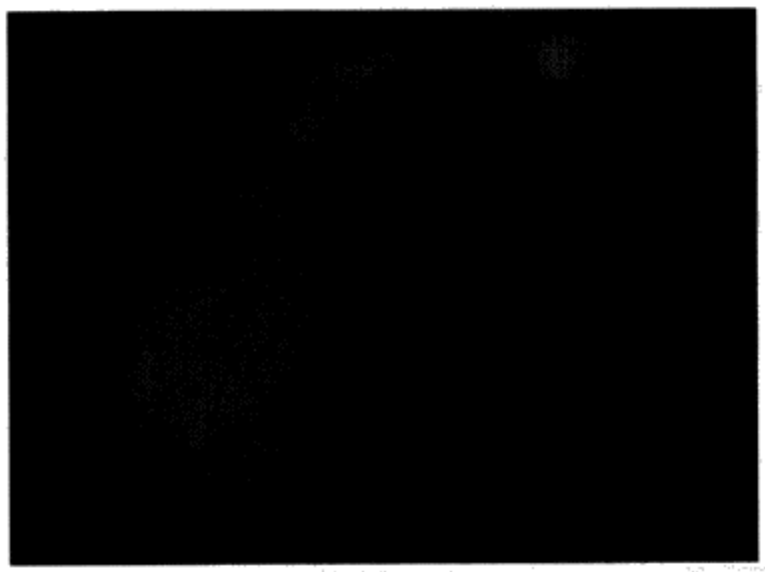


图 7-3 “18 英寸。就在这儿。写明了的。”



是在规格说明里写明了的！或者，根本没在规格说明里写出！这就是每个遭遇阻碍的软件项目不可避免会发出的重复呼声。编写规格说明——用来指导程序员工作的细节丰富的文档——对于那些最终用户不

是程序员的软件企业来说是必须环节。规格说明将需求——软件开发者的客户提出的目标或愿望——翻译为指挥程序员的详细行军指令。例如，对于一种个人财务软件产品，需求可能类似于这样：*必须支持多个信用卡账户的分类账目。要能从银行下载账户信息。*规格说明将明确写出程序**如何**满足这些需求。界面规格巨细靡遗地描绘出每个屏幕的样子，按钮和菜单的位置、字体和颜色，以及当文本超出一行时该如何处理。交互规格说明记录程序如何响应用户每次点击、拖拽和字符输入。

规格说明是程序员的圣经，而且，通常程序员也会是忠诚信徒：规格说明就是法律。因其天性和职业需求，程序员也缺乏想象力。所以编写规格说明就要十分当心：像在童话故事里一般，你要小心阐述自己想要的东西。（永生？别忘记还要指明“青春常驻”！）

计算机界有个老笑话，说某位受尽挫败的用户拨打技术支持热线，抱怨说明书里写“按任意键开始”，可怎么也找不到“任意键”在哪儿。这个笑话嘲笑的是那个倒霉的白痴用户，但这新手的错误理解却真实反映出关注细节、无视上下文的阅读方式是编程大牛们的专长。规格，顾名思义，应使用详尽的特性描述、在人类模糊世界与机器精确世界之间搭起一座桥梁。不该让程序员如坠云里、连猜带蒙。但这类努力不免有失完美。部分原因是规格说明的作者是人类，他们用人类语言写作。但也因为写作“完美的规格”——设想并规定程序使用和行为的每种可能场景——耗力无算；永远不可能在完成规格撰写后才开始编码。

实际上，人类的耐心缺乏再加上市场需求迫切，常常导致规格说明编写不充分。而在 **OSAF**，问题却不在于知道规格说明何时完成，而是构想出如何开始写。撇开团队内部对动力的新感觉不谈，在几个关键且复杂的部分仍然盘踞有“蛇”——用 **OSAF** 早期的术语来说。

共享信息是 **Chandler** 最重要的特性之一，但每次设计组讨论到这个主题，总会陷入未解决问题——或者用项目管理语言来说，“开放性问

题”——的循环中去。用户如何发起共享？到底可以共享什么——单个条目？整个条目集合？还是一种特定的视图或一组屏幕布局？当用户要共享这些而不是那些条目时，Chandler 如何才能控制“权限”？如果允许某些共享者修改共享的条目而另外一些共享者不行，如何才能让数据拥有者做出选择？如果有两个用户在离线时修改了同一条目，当他们回到线上、试图同步修改时，Chandler 应该如何解决冲突？跟着是最痛苦的共享问题：“链式共享”。如果你共享了我的一个条目，你能和第三人共享这个条目吗？她能再与别人共享吗？当第三位、第四位、第五位共享者修改了这个共享条目会发生什么——修改会沿共享链条传递回来吗？该问题又名“*n* 向同步”——同步数据的未知（*n*）个版本。

一开始时的简单需求——“用户应该能方便地共享信息”——很快扩张成一大团难题之云。虽说暂时可以推到“以后再说，”但每个人都知道，最终还得设法解决这些问题。

共享并非唯一难点。围绕 Chandler 群集（或条目组）的管理还有许多开放性问题。从某种意义上说群集类似桌面应用界面中的文件夹——包含一组东西——但从另外的角度来看，它们更为奇特。条目可能同时在多个群集中存在。有些群集基于规则：只要让 Chandler 创建一个包括所有商业伙伴名字的群集，程序就会自动更新群集。在这些部分，Chandler 管理信息的方式借鉴了 Apple iTunes 软件管理数字音乐集的方式。但卡普尔也想确保 Chandler 可以允许随意剔除或包含条目。他常举例说，想要一个基于规则的、包括所有鲍勃·马雷（Bob Marley）作品的音乐集——但要剔除一张他很不喜欢的现场版专辑。用 iTunes 可做不到。

“第一次，”卡普尔在一次会议上解释说，“在设计 Agenda 时，当我说到我想要用规则定义群集、但也会有随意分组的情况的时候，遇到了极大的阻力。就像是我犯了计算机科学中某种大忌似的。没错，有许多

非常疯狂的情况。但不能因为我们以为人们会害怕就放弃特性。这样你就永远发明不了蹦极绳。”

不过，为了让 Chandler 能处理卡普尔的鲍勃·马雷个案，OSAF 设计组要琢磨许多事：用户到底会如何创建类似群集并详细说明包含、剔除条件？Chandler 如何存储这些群集？怎样才能保证人们不会搞混从群集中剔除一个条目和彻底从 Chandler 中删除这个条目？在 Chandler 界面上如何显示所有这些不同类型的群集？

还有就是被称作“条目易变性”或“多态性”的问题。Chandler 的使命是消除其他个人信息管理系统用来隔离不同类型信息的“地窖”：电子邮件、日历事件、待办事项和便签都在同一个大信息池中共泳，等待用户按需浏览。但到底用户会怎样获取一封关于派对的邮件并转换成日历上的事件——或者把从同事处收到的邮件转换为待办事项列表上的任务呢？这封邮件会从邮件群集中消失还是保留下来？最后会得到一个条目还是两个条目？

在设计组全力解决这些问题时，卡普尔说起一句编程格言（普遍认为来自个人计算机先驱艾伦·凯）：简单者就该简单，复杂者亦有可能。卡普尔希望 Chandler 易于使用；他也希望它能解决一些著名的软件难题。现在，他和他的队员们深深陷入了折中的困境，两者都想要。

2004 年 4 月的一天，林智超给尹咪咪发了一个指向在线通讯动态领域资深评论家克莱·舍基（Clay Shirky）blog 帖子[①]的链接。舍基在文章中说自己重新发现了克里斯多夫·亚历山大——那位建筑哲学家，他关于“模式”的概念被借鉴到编程世界并迅速走红——的一篇旧文。这篇写于 1965 年的文章题为《城市不是树（A City Is Not a Tree）》，作者观察到规划社区通常被设计为“树状结构”，分析了规划社区的失败：“只要有树状结构，就意味着在结构之内任意单元的任意部分都不与其他部分相连，除非是以所属单元为整体媒介。这种限制之巨大难以估量。有

[①]这个帖子的地址是 http://many.corante.com/archives/2004/04/26/a_city_is_not_a_tree.php。

点像是家庭成员不得自由与外人交友，除非全家人皆与其订交。”

用亚历山大的术语来说，有机增长的真实城市——以及人类关系的真实结构——是以“半格点”形态存在。半格点是比树状更松散的结构，仍有继承层级，但允许子集重叠。为什么建筑设计和规划社区总是“树状结构”呢？亚历山大认为，半格点更为复杂和难以描绘，而且我们不可避免地倾向于采用更易于把握的树状结构。但是这种“每个思维简单的人都患有的将同名物体放入同一篮子的狂躁症”却在城市设计中导致了人为的约束和隔离。“采用树状结构，就是以人性和鲜活城市的丰富性为代价，去换取概念上的简明性，这只便利了设计师、规划师、行政官员和建设者。每当城市被撕开一块，用树状结构替代了原先的半格点结构，城市就向着分裂又迈进了一步。”

尹看了亚历山大的文章——文章通过克莱·舍基的 blog 和林智超的邮件传递给她，这条路径似乎正展示了文中关于复杂联系的观点——而且立刻把文中提到的概念应用于 Chandler 的设计。她为 Chandler 浏览器部分做了精细的设计，稍稍借鉴了 iTunes 软件的设计。用户通过混合与匹配（或“切片与划线”）三列选项（例如，第一列选中“电子邮件”，第二列选中一组名字，第三列选中日期范围）浏览资料库中的条目。

浏览器的设计方案引发了冗长的讨论，但没有赢得一致同意。在尽力为 Chandler 信息设想一种层级更少的不同架构时，尹首先了解到，层级结构已深深嵌入到软件本质和软件创建者们的思维之中。如亚历山大所言，城市可以不是树状的，但今天几乎每种计算机程序都实实在在是棵树——一种由代码行组成的层级结构。在软件世界中到处都有树——从在许多程序用户界面左栏普遍存在的文件夹树，到文件系统和数据库的深层组织形式，再到开发者用来管理代码的系统均是如此。

“树”是开发者签入其工作成果的源代码目录的非正式说法。假如某个小组要对程序一部分做单独开发，又不想打断其他同事的工作，就要

从树的“主干（trunk）”上分出一条“分枝（branch）”。而后，在打算重新集成到其他代码时，新分枝就和主干“融合（merged）”了。每当像OSAF的新Tinderbox这样的测试系统报告说项目一切正常，程序员们就会如释重负地说：“树还是绿的。”

所以抛开树状结构大概不太容易。无论怎么说，都感觉尹的浏览器也和应用组开发者们的眼前问题和需求没什么关系。要实现类似的东西，他们还差得远。他们面对的下一道巨大障碍是实现Chandler的“细节视图”。Chandler屏幕分为四个基本区域。顶端横贯着传统的“镀铬区（菜单、工具条等）”。左边是开列和组织条目群集的“边栏”。中间是主要区域，分为一览表——那个“大表”曾是贯穿整个2003年的绊脚石——和细节视图。一览表将列出条目（电子邮件、日历事件、用户输入Chandler的任何数据）；用户在一览表中点击某个条目时，细节视图中就显示关于该条目的全部内容。

这种结构在传统PIM和电子邮件程序中都很常见，但仍然做需要大量工作来考虑到底如何设计和构建每个元素。一览表视图尽管离完成还远，到底是在进行中了。而细节视图则成了眼前最大的障碍。它不止要编目和显示与条目有关的文本；Chandler消灭地窖的野心意味着开发者们也要面对全新的问题。拟出视图包括、显示什么以及不包括、不显示什么的简单工作已经成为决策的焦点——卡普尔喜欢称之为强迫性功能。创建细节视图、将这些选择的结果转化为代码的工作落到了OSAF的新程序员唐·登曼（Donn Denman）肩上。



有些程序员在乌云中彷徨，仿佛自己修正过的每个缺陷都在他们的心灵上留下了一道伤疤。另一些程序员则视编写代码为唯一乐事。唐·登

曼就是第二类程序员。

“我年轻时就爱玩解密游戏，”我问登曼怎么会干起这行时，他这样告诉我。“我真的超爱玩穿绳之类的解密游戏。现在如果见到设计得好的谜题，我还是会花好几个钟头琢磨怎么把绳环从矩形块里面解出来。即便看起来无解，也要琢磨到底怎么回事。”

“读书时我还喜欢上数学课——我想，这类东西之所以能吸引我，是因为当我做对的时候，我**知道**自己做对了。在解数学题时，我常先给出答案，然后再回头运算，并认定那就是正确答案。”

“对于没有缺陷、运行正常的程序也是如此：你可以看到自己的成功，而生活中好多其他事却太主观，你不知道是否能成功。这对我来说不太安全。而且，我真的很享受能够创建、制造、拆解物品并查看其工作机制。还有修理。现在我觉得这类客观性奖励没那么重要，我长大啦，更加自信，不需要那种反馈了。但创造点什么、看见事情发生、制造东西——还是很有趣的。”

登曼在俄亥俄州的黄溪泉（Yellow Springs）长大，安提奥奇学院（Antioch College）[①]就坐落于此地。他父亲是学院教授，他自己也从这所学院毕业。1979 年，他一毕业就到苹果公司工作，办公室就在安迪·赫兹菲尔德的办公室正对面。在这家年轻的公司里，他发现一种让自己想起安提奥奇学院文化气息的反叛精神。登曼总是醉心于创造让没有或只有一点编程经验的人们也能用计算机自动化其工作的系统。在那个时代，Basic 就是做这类事的工具。在苹果公司，他起初只负责将 Apple II 版本的 Basic 移植到 Apple III。

后来赫兹菲尔德邀请他进入麦金塔团队——他的工作还是为新计算机编写 Basic 语言，但要体现 Mac 机革命性图形界面的优点。“我们有大把时间，”登曼回忆道，“而且我也知道如何从头做一套新的 Basic。我工作有进展，不过却无法真的给项目设限，使其按时完成并和第一代 Mac

[①]一家历史悠久的私立艺术学院。

同时推出。”



图 7-4　存储 AppleSoft Basic 的磁带

登曼一直在做 MacBasic，但在完成之前，微软就发布了自己的麦金塔 Basic——苹果程序员视其为等而下之者、与 Mac 新设计整合不佳。当时，苹果与微软关于授权在 Apple II 上运行 Basic 的合约到期。比尔·盖茨知道苹果亟待续约，作为条件，微软要求苹果停止登曼的 MacBasic 开发工作。苹果公司把代码以区区 1 美元的代价卖给了微软。

安迪·赫兹菲尔德在 Folklore.org 网站上讲述了这个故事，“当唐得知 MacBasic 项目被取消时，他心碎欲绝。经理告诉他，‘项目无限期搁置’，建议他销毁所有代码和副本，但拒绝告诉唐到底发生了什么事。那天稍晚时候，唐骑着摩托疯狂飙车，把车撞了个稀烂。好不容易回到家，身上倒是没伤，只是那点自尊心已被彻底敲碎了。”

“我花了那么多时间和精力在它上面，它就像我的孩子一般，”登曼跟我回忆说，“两岁大的孩子就这样被带走了。我花了 10 年或 20 年的时间才熬过来。”最终他留在苹果，为另一种给非程序员准备的系统 AppleScript 工作。20 世纪 90 年代，当苹果公司失控下跌，他离开了公司，为被称作机顶盒的数字有线电视分发系统编写软件，度过了技术工业泡沫时期。泡沫没让他发财，但也留给他足够的钱，可以稍稍休息一

下。他喜欢用自己的时间做自己的项目，但当赫兹菲尔德开始告诉他 Chandler 的事时，他两眼放光。他一直渴望重拾旧爱——为非技术人群做脚本系统。卡普尔的项目听起来就是那张入场券。

登曼于 2004 年 4 月加入应用组，和凯蒂·帕兰特一起考虑具体做什么。负责细节视图似乎比较便于让他深入 Chandler 代码、了解 CPIA 的特性。他将是第一位没参与创建 CPIA 却要认真使用它的程序员，而他提出的问题将有助于把 CPIA 做得更好。

为了搭建一个能工作的细节视图雏形，登曼需要一份规格说明。看来现在还没有那东西。尹咪咪已经有一大堆 wiki 页面和细节视图界面的粗糙框图，但设计组会议上持续提出的改进建议却不断抛出关于共享、群集和条目多变性的问题。登曼意识到得和设计师们打个商量，开始编写自己的规格说明。所以登曼、尹和林智超在五月份几乎每天在沃德街的小会议室——每间小会议室都没有窗户，刚好能容纳三、四个人，还有一张桌子和一具铺满整面墙的白板——中开会。过个把小时，屋里就热起来了，干性马克笔的气味充盈四周，但小组坚持要趟过所有开放性问题，为登曼清出一条开始编码的道路。

尹阐述了关于条目的最新考虑。多数 PIM 程序都要求用户在创建新条目之前就决定是什么条目：你要创建一封新电子邮件吗？一次日历事件？还是一个待办任务？Chandler 应反其道而行之，让你可以安坐椅中、开始输入笔记，稍后再决定写的是什么类型的条目。笔记就像人体内无差别的干细胞，有可能往不同方向生长，开始新生命。这种设计旨在将向程序输入信息的基础行为从监禁地窖中解救出来。它也给尹的条目多样性问题解决方案留下了空间：用户指定条目类型的机制可以被叫做“打戳”。

比如，你输入一条笔记——关于某次会议的一组句子。然后你想在日历上加入这次会议：只要给笔记打上“事件”的戳就好。Chandler 的

细节视图会添加几个栏位，让你指定日期和时间；普通笔记现在变成了一次事件。稍后，假使你打算邀请某位同事到会，可以给同一条笔记打上“电子邮件”的戳：细节视图上会出现“收件人”和“主题”的栏位；填好这些空栏，点击“发送”按钮即可。

打戳是个好主意，但它引发的问题和它回答的问题一样多。能取消笔记上的戳记吗——如果这么做了，与被移除类型有关的信息会怎样（例如日历事件上的日期信息）？当你给一条笔记打上戳，再查看“全部”群集（列出 Chandler 资料库中全部类型的所有条目）时，会看见条目的两个副本吗？

尹着手描画细节草图，登曼仔细看过后问了她几个问题：这个标签是什么意思？那个特性如何运行？

5 月的一个下午，设计组和登曼一起努力完成细节视图的第一轮规格说明，他们开始往白板角落仅余的空白处写东西。

“我需要显示新条目，不过还没想好怎么搞。”尹说。

“这是什么？”登曼指向草稿上靠近细节视图顶部的一个小人头图标。那是共享功能的按钮。尹解释说——用户点击这个按钮即可将条目发给别人。她很怀疑这个图标是否该在电子邮件界面显示，电子邮件应该有自己的发送控件。“可如果电子邮件变回任务项，就需要它了。”登曼在笔记簿上划拉了几下。尹停下来，转身背向白板，说，“这些就是让我做噩梦的事情啊！”

“嗯，”登曼抬起头来，“我来帮你搞掂吧——我们创建一个子集，看看怎么好办。所以，至少在 0.4 版之前就根本不用那些小人头了。”

“不，”尹答道，“还得要——否则，就共享不了。”

“那就别共享了。”登曼微笑起来。他可能在开玩笑；也可能不是。

尹在电子邮件里草草写下白板上的内容，记录他们的讨论和决定。

电子邮件就这样变成了多数计算机用户保存各种信息的仓库——这事多数电子邮件程序做得很糟糕，搜索功能极差并拥有脑死一般的附件管理功能。正如 **OSAF** 开发者们常说的那样：“我们真能用 **Chandler** 干这个。”

“我要花一个来星期做计划，希望接下来进入实现阶段，”登曼说。他补充道，“新工作还是挺有趣的！”——他稍稍微笑了一下，说明这是种“半空还是半满”的乐趣，而且他积极地选择从“半满”的角度看问题。

“感觉就像早期在苹果的日子，”在实现打戳功能期间的一个晚上，登曼告诉我，“也是我对这份工作感到兴奋的原因之一。真的感觉像是我们能改变这世界——当然，尽管我们在这里和苹果都没讲出口，但我们确实一直在这么说！”



打戳机制旨在给计算机桌面引入一种富有生产力的多义性，更接近地反映出人类的思维方式。这不是个简单概念，即便对于创造它的设计师来说也是如此；对要使之工作起来的开发者而言，甚至更加需要技巧。计算机程序采用地窖、树状结构和类似的明确结构，是因为这些结构有助于保持组织数据、减少混淆。要是一个条目属于某组，就不该归到另一组；要是它在树的一条分枝上，就不该在另一条分枝上。

人类语言的容错性较强：一个单词可以表达多种意思。这种灵活性打出了微妙与美好的深井；那是诗歌的基础。但在搭建软件时却会导致麻烦。当 **OSAF** 的开发者们致力于在 **Chandler** 中引入类似打戳之类的创新时，从草案到功能性代码，他们不断发现自己被多义性绊住。一次又一次，他们用相同词语描述不同事物。

以*条目（item）*为例。对于设计师而言，**Chandler** 中的条目可以是

用户设想的任何基础数据片段。一封电子邮件。日历上的一次事件。一个任务或一条笔记。但在 Chandler 资料库的后端世界也有条目，它微妙而充分地与前台条目区分开来。资料库条目是 Chandler 数据库中存储的一块信息，事实上你需要好几个这样的资料库条目才能向 Chandler 用户展示类似电子邮件那样的一个用户条目：用户条目的每个属性——主题行、发送日期、发送者地址等——都是独立的资料库条目。在 Chandler 演进的不同阶段，提出过多种解决此项难题的方案——让*条目*一词“不具多义性”。也许术语“用户条目（user item）”一词该一直用大写形式。（在人们记得这么写时，对文字材料有用，但在对话中没用。）也许应该给每种条目取个名字。（不过这些名字本身也许更具多义性，例如 *thing* 一词，所有这些名字都够呛。）

Chandler 的宇宙中充满了这类词语重叠。设计组一直采用术语“*数据模型（data model）*”指称用户可能遇到的数据类型定义列表，以及与数据类型有关的所有属性。例如，数据模型可能指明，每条笔记都要有“创建日期”、“作者”和“正文”。但对于开发者而言，*数据模型*是指一种更技术化的不同定义集，用来在代码层保存和取出数据。设计组所说的数据模型其实是指开发者所说的*内容模型（conent model）*。

术语“*计划任务（scheduled task）*”也有问题。在设计世界中，计划任务是指在用户待办列表上指定了日期和时间的一个条目。但对于开发者而言，计划任务是指 Chandler 被告知在特定时间要执行的操作，比如下载电子邮件，或者在共享信息中检查更新。又比如术语“*事件提醒（event notification）*”。对于设计师而言，它表示告知用户有新邮件到达之类的事情；对于开发者而言，事件是代码对象状态的改变，比如窗口关闭或在菜单项上点击——而提醒则表示向其他代码对象发送改变的信息。

如果卡普尔留意到这些语言学列车的小小出轨事故，“咱们要说同一

种语言，”他会这么说，“我们要说 Chandler 语。得修正词汇表才行。”

最后，布莱恩·史金勒挺身而出接手此事。在维达刚开始做资料库时，史金勒作为志愿者加入 OSAF，帮助凯蒂·帕兰特整理数据模型的细微之处。史金勒现在是 OSAF 的全职程序员，他擅长向设计师解释开发者的语言，以及向开发者解释设计师的语言。当两帮人马自说自话，他总能设法为之沟通。他建议说，何不在 wiki 上建立一个 Chandler 术语表呢？那将对围绕项目的所有术语提供单一、权威但易于修正的参考点。它将能在文字上统一术语。

史金勒着手对抗多义性的海洋。他创建了成打的术语表页面。他还构建了一套系统，把这些术语页面链接到 wiki 上的其他页面。

这是英雄般的成就，但没能改变什么。首先，术语用法的变化速度仍然快于他编辑更新 wiki 的速度。更重要的是，那些已经浸没在电子邮件、缺陷报告和 wiki 页中的开发者们，看来并没有太关注术语表，多数页面无人阅读，逐渐枯萎了。

术语表流于形式，这可能早已被史金勒参加的另一次命名行动的结果所预示。OSAF 在沃德街的总部有 6 间会议室，卡普尔认为给它们取名字会有用，这样就不用说“咱们到唐办公桌那边角落的小会议室开会吧”，只要说会议室的名字就可以了。OSAF 在员工中举行了一次征名赛。史金勒提议用幻想出来的地方命名；他捧得桂冠，得了 150 美元的奖金，和同事们一起到街头的爱尔兰酒吧 Kate O’Brien’s 快活了一番。

那些天马行空的名字——两个大会议室分别命名为阿瓦隆（Avalon）[①]和阿卡迪亚（Arcadia）[②]——撷取了 OSAF 的精神，相信新世界建立于团队共同的待办任务列表上。到后来大家才意识到，最常用会议室的名字都以字母“A”开头是个糟糕的点子：谁也记不住哪间是哪间。

[①]凯尔特族传说中的西方乐土岛，据说亚瑟王及其部下死后尸体被移往该岛。

[②]本为希腊一省，因其多山而少居民，自古被当作世外乐土的代名词。



在每个程序员网站上，而且我敢打赌在每家软件公司的小办公间墙上，都贴着一套《呆伯特》漫画。漫画的主角是世界上最受欢迎也最受压迫的奇客，他对主管（著名的蓄双尖发的老板，即 PHB）说："我们还是有太多软件错误。赶不上发布期了。"PHB 答道："把出错列表移到'未来开发计划'栏，然后就发布吧。"在最后一格漫画上，PHB 自我安慰，"我这职位 90%的工作是琢磨怎么给事物命名。"



如奇客们常说的那样，程序员给他们抽象出来的东西取的名字"非比寻常"。在软件中，标签和名称很重要；它们是抓取事物的把手。

婴儿在把自己的词汇表扩展到"妈妈"、"爸爸"之外时，是通过将声音和可见的物理对象联系起来而学到的；我们指向某样事物，重复其名称。教会孩子更抽象的词语的意思就难多了；我们得用另外的词来解释抽象词，或者讲个故事来说明。

在用户界面的窗体和文本之外，软件的大多数元素无形无质、不可目见。没什么可指的。所以谈论这些东西也出乎意料地难。这就是在软件开发中白板无处不在的原因之一；它为描绘复杂程序的抽象过程提供了画板，让人们在讨论时有物可指。

在做新程序时，程序员的第一个任务就是为程序使用的变量和对象命名。程序用变量在计算机内存中预留一块地方，可以往里面放置信息。稍后，可以根据其地址——通过呼叫其名称——从中取出信息。

机器内存要求在变量名和内存地址之间存在一一对应关系。与之相反，人类大脑却层次丰富、互有重叠；其密度及无边界性造就了隆起之

外观、碰撞出直觉之火花。但大脑的特点在数字机器里却变成了缺陷。拜错误或糟糕的编程所赐，若两个程序企图在计算机内存中同一位置保存变量，必会出现不可逆料和不稳定的结果（通常导致计算机死机或崩溃）。软件痛恨多义性。

在从上下文中找出线索、梳理出多义符号最可能的意思上，人类远比计算机能干得多。然而如果要想让计算机理解我们，通常还得先理清含混不清之处。即便面对一台擅长手写识别的计算机，也不要问它草书字母“O”和阿拉伯数字“0”之间有何不同。（老式计算机终端常在数字 0 上加一斜杠以示区分。）在计算机文化有发明它的需要之前，单词“*disambiguate（消除歧义）*”并不存在。

当程序员们坐到电脑前，开始进行为程序变量命名的圣经式操作时，他们必须保证变量名的词义清晰而独立。在“我猜想……”的粗糙原型阶段，许多程序员使用 *foo*、*bar* 这两个万金油变量名。这两个词的来源仍有争议，但多数人相信它们派生自历史悠久的军队缩略语“FUBAR”（“Fucked Up”——或“Fouled Up”——“Beyond All Recognition/Repair/Reason/Redemption”）。这些无意义的占位符迟早要被真正的变量名所替代，例如 AccountHolderID 或 LastDateModified。

谨慎取名才能避免“名称空间对撞”和“冲突”的混淆状况——某个术语在一处上下文表示某事，而在另一上下文则另有含义。Wiki 发明人沃德·坎宁汉甚至建议说，程序员应该在手边常备一本同义词典。他在 2004 年的一次谈话中强调了取名的重要性，“因为**别人会读到**这些名称。你知道，人家会猜想，对象就是其名称所指的那样。不过，通常变量都不是其名称所指的那个意思。”当程序员开始工作时，名称是这个意思，在修正了一千个缺陷后，却变成了其他意思。坎宁汉鼓励程序员回头修正名称——但他也承认多数人不会自找麻烦。

对变量和软件对象的命名做严格要求的历史，几乎和计算机编程史一样悠久。在早期，计算机内存中的可用空间相当有限，优秀的程序员

们学会了通过缩写变量名来节省空间。用“memloc”代替“memory location”来命名，这的确有很大不同——特别是当变量名在程序代码中重复几百次的时候尤为如此。

最著名也是最声名狼藉的变量命名方案是匈牙利语标记法。其名字来自于发明者查尔斯·西蒙尼（Charles Simonyi）的母语名，同时也来自该种语言神秘而不易发音的单词标记，西蒙尼在微软的同事认为这些标记具有暧昧的外国特征。在匈牙利语标记法中，程序员给每个变量名加上前缀，好让代码阅读者了解变量的类型。西蒙尼在施乐做研究员时，开发了第一个图形界面的文字处理器，后来他又成为世界上最广泛使用的书写软件 Microsoft Word 之父，而他的匈牙利语标记法约定也成了微软的惯例。多年以来，匈牙利语标记法的拥护者辩称它使代码具有可读性，而其反对者则认为它把代码变成了八股文。稍后时加入 OSAF 的网景程序员阿列克·福莱特（Alec Flett）曾经在邮件列表上这样恶搞：

prepBut nI vrbLike adjHungarian! qWhat's artThe adjBig nProblem?[①]

鉴于命名约定的目的是让代码编写行为更为清晰，看来极具讽刺意味的是，在西蒙尼首次撰文解释时，该种命名法也饱受多义性之害。匈牙利语标记法后来成了**两种截然不同的东西**。开发 Microsoft Windows 的程序员们所用的匈牙利语标记法，后来被称为系统匈牙利语（System Hungarian）标记法，和西蒙尼最初向 Word 程序员们宣传的匈牙利语标记法有所不同，后者现在被固执人士叫做应用程序匈牙利语（Apps Hungarian）标记法。西蒙尼说变量应根据其“类型（type）”命名，系统匈牙利语人群理解为多数编程语言要求你为变量选择的那种“类型”（整形、浮点型、字符串型）。但应用程序匈牙利语——所谓**正宗的**匈牙利语标记法——则认为“类型”的意义更为松散、更接近日常用法，鼓励你在变量中加入更多可以真正对阅读代码的程序员有用的沟通信息。

一路走来，微软热心的程序管理人员们误解、忽视或抛弃了西蒙尼

① “我就是喜欢匈牙利语（标记法）！有什么问题？”阿列克在每个单词前面加了其词性的缩略语前缀，以此嘲笑匈牙利语命名法的前缀要素。

的本意。结果，匈牙利语标记法——借用开源程序员们用来形容分裂项目的说法——分叉了。所以虽然你用匈牙利语标记法编码，我也用匈牙利语标记法编码，而我们的代码却可能永远不能统一。



没有歧义地沟通抽象概念——程序员和机器之间、程序员和程序员之间、程序员和用户之间——是软件开发中最具挑战性的任务。程序员和非程序员如何才能避免巴别塔之灾？有些代码编写者苦练用极度精确的语法与机器对谈的功夫，却丧失了设身处地与人沟通的简单本领。如果把这类程序员关到一起，要他们就某个词达成一致，否则不准出来，结果极有可能是再也见不到他们了。

OSAF 开发者们在程序员光谱上基本都处在更具社会沟通能力的那端，但他们仍旧时常被“同一单词，不同意思”的问题所绊倒。如果你真的够运气，大概能找到一、两位富有想象力、好脾气和愿意聆听的开发者，他们也许能在产品开发或设计方面与你志趣相投，也许能一起剪除多义术语的杂草、清理出一片空地、解决某些难题。在 OSAF，当唐·登曼和尹咪咪共同研讨细节视图规格的每个细微之处、探讨每个小图标和交互特性、直至登曼可以开始编码时，类似的事情发生了。



5 月末的一天，登曼把尹咪咪拉到办公桌前，指了指计算机屏幕上一些彩色矩形的布局安排。如果去掉那些文字标记，看起来就像是一幅抽象画。他们开始讨论怎样构造“条目对话”特性的界面，这个特性允许用户给每个共享条目添加简短的注释。尹突然停下来，盯着屏幕。“等等——你是说，现在就能运行？”

“对，是在 Chandler 里面啊，”登曼答道。

“酷！我还以为就是一张图呢。”

“哪有，我喜欢在这儿看到东西！一开始我搞了个普通布局，然后把元素连接起来，现在这东西真可用，按钮也可以点击。”

一周后的周四会议上，登曼向整个 OSAF 团队展示了这个场景。“这些是占位符，”他说，“还不是已经做好的东西，而是要做的东西。暂时我们只能让这堆东西可以工作、但不太好看，或者看起来不错、但不能工作。”

接下来的一个月，登曼和资料库程序员安蒂·维达以及泰德·梁合作，想出了扩展资料库、支持打戳功能的方案。尹和设计组主要把精力放在设计打戳最可能出现的情况上——把收到的电子邮件戳记为日历事件或任务；把普通笔记戳记为电子邮件、日历事件或任务。但程序员总会考虑“边缘情况”，而登曼及其同事编写的打戳系统最终也会是一种“通用解决方案”——试图允许你给任意类别的任一条目打上任意其他类别的戳记。他们提出了一种称作“混入（mix-in）”的概念。混入定义了与某一类别有关的一系列属性：对于待办任务，可能包括“优先级”和“完成状态”。如果把一个电子邮件条目戳记为任务，资料库就会简单地把新的属性集加到邮件原有属性集（如“发件人”和“主题”）。这样就难以避免出现“名称空间冲突”：“日期”这个简单属性在电子邮件场景中可能是这个意思（收到日期），而在任务场景中却可能是另外的意思（到期日或完成日）。如果老板给你发邮件通知与客户的会面，你把邮件戳记为日历事件，邮件中的“谁”（你老板）与事件中的“谁”（客户）全然不同。Chandler 的后端能分清这些日期和“谁”，但程序还是要择其准者显示给用户。登曼的方案没有解决所有问题——只是能够让它们开始干活而已。

7 月初，登曼和应用组的几个开发者围坐在阿卡迪亚（或阿瓦隆？）会议室的投影仪旁，复查登曼的打戳功能代码。术语“*代码复查（code review）*”既可以是在计算机旁的非正式讨论，也可以是长达数周的衙门

过堂般层层剥茧的代码检视。OSAF 通常总是选择较为轻松的复查方式，但开发者们最近都同意做更多代码复查，而打戳功能是如此新颖而复杂，的确值得关注。

登曼先概括“混入（mix-in）”的概念，再回顾了一些讨巧的处理手法。“接收电子邮件时，最重要的日期是接收日期。戳记为事件时，最重要的日期是事件发生日。所以我耍了点手段来处理。”他向下滚动屏幕，投影仪投射出另一段代码。“抱歉，真难找。”又滚动了一下屏幕。“我改了 Self 类。我在这儿扔一个资料库事务[保存日期状态]进去，这样我们就会记得有个新类别……如果能够猜出新类别是什么，我就指定这个类别。否则，就用安蒂的混入类别。”屏幕上下一段 Python 代码定义了 NewStampedKind。“这段置入代码基本上用来在混入后查看是否有与那堆组合属性相匹配的类别。”

我装不出能全听懂登曼阐述的样子。门外汉参加代码复查，就像是偷听专家说行话；你无法每次都能抓住问题的症结所在。但我还是能明白，登曼让 Chandler 更敏捷，并解决了打戳的问题；现在，Chandler 可以简单地通过查看条目的属性集来判别条目类别，而且还能在不丢失条目在原类别中具有属性的前提下转换到新类别中。

我也了解到，登曼认为还有很多工作要做。

登曼说完后，约翰·安德森抬起头来。“我得花点时间消化。了不起，只是有点复杂。我的直觉说咱们能搞掂。”

实现手段很复杂。但它具备最重要的优点：一周后——大约在登曼开始为 OSAF 工作后十个星期——打戳功能可以运行了。也许只是首轮通过，可能还会遇到障碍，但它毕竟是运行起来了。登曼在接下来的周四员工会议上做了演示。

“简单直接，”他一边说，一边在新的细节视图上点来点去。“现在可以在打戳时改变条目类别。”他在 Chandler 中显示一条标记着“梦想药

方”的略记。“刚开始是条笔记。点这个任务小图标就能将其标记为任务。可以看到，有些属性改变了——现在有了任务发出者和接受者栏位。目前我们可以打三种戳——任务、日历事件和邮件对象。也可以三者混用。理论上，只要有内容条目的新类别出现，都可以照打不误——可以给相片打上‘邮件’戳，相片就会有邮件的属性，能够被发送出去。”

“条目可以改变类别，这是基础概念。大多数工作不归我做——都是安蒂在资料库里面的事。条目不改变 ID，还是同一个条目，只是发生了演化。当改变类别时，它就会查找是否有匹配该类别的属性。如果有，就应用这些属性。如果没有，就自动创建一个新类别。”

演示在雷鸣般的掌声中结束。卡普尔微笑着发表了讲话。除了共享，打戳就是 OSAF 开发路上最大的拦路蛇了。它即便还没被彻底干掉，但也算是乖乖服管了。地窖都塌啦！但还有远路要走。

“与打戳有关的特性贯穿整个 UI——我们可能只实现了百分之一，”卡普尔说，“完成这个后，将会是有别于其他 PIM 的独有特性。条目进来，你点击，条目显示在适当的位置。那将是另一番境界——人们会说，当然，就该这样！为什么别人的东西不是这样呢！成功与否取决于咱们为人们重构该问题的优秀程度。”



最近，卡普尔给自己提了个更个人化的问题：他应当多深入 OSAF 的事务？他该为 OSAF 做什么？

自迈克尔·托伊于 2003 年 11 月离开以来，他不仅是产品愿景的监护人和 Agenda 之魂的守卫者，还担当了设计及开发经理的职责。一开始，当团队还在 0.2 版失败带来的阴郁中举步维艰时，他发现这个角色令人精神爽利。但几个月过去了，工作量持续增加，特别是新的开发者加入后更是如此。曾想做乐师的年轻程序员布莱恩·科斯齐（Brian

Kirsch）参与编写 Chandler 的电子邮件代码。苹果公司老手戴维·萨鲁维（David Surovell）将负责为 Chandler 改进 wxWidegts，特别是在 Mac 上的运行。开源日历标准领域的领导者丽萨·杜索特（Lisa Dusseault）将和凯蒂·帕兰特一起管理不断增长的程序员队伍。

要求卡普尔出席的会议越来越多，而他在这些会议上的发言也越来越频繁，声音越来越大。他不止一次做出“暂停”的手势。“对不起，我现在缓冲溢出，”他承认，而屋里的每位奇客都明白他的意思：*事情太多、超出了我的处理能力*。他不是那种爱发脾气的老板；这让他与业界多数大腕区别开来（比尔·盖茨的怒气人所共知，史蒂夫·乔布斯也是如此）。我在 OSAF 的三年里，只有几次听到他大声说话，而且总是跟着就马上道歉。但在 2004 年春天，他的耐性被磨灭殆尽。

与此同时，他的注意力也被 2004 年大选吸引到另外的方向。打败乔治·W·布什的自由派希望正在高涨——推动者是由 MoveOn 等新兴在线社区、blog 和组织所组成的新生筹募组织力量。卡普尔对“数字化民主”有长远的看法，但他也要抓住当下：他创办了名为 Of, By and For[①] 的政治 blog 群，还帮忙推出了搭建 CivicSpace 在线社区组织的开源项目。不管大选结果如何，软件都会继续运行。

在很长一段时间内，卡普尔就像 OSAF 的其他许多同事一样，习惯于每天花几分钟时间到 wiki 上看看有什么改动。要看到这些改动，可以访问网页、接收邮件列表，或者通过 RSS 源获取信息。系统不算完美——有人更新了整页或是只修改了一个逗号都会报告——但却是紧跟 Chandler 团队日渐扩大分散的好法子。然而，在 5 月中旬，当团队重拾进度、wiki 上一页页记录下唐·登曼在细节视图上的工作时，卡普尔突然感到自己不堪重负了。

他因为相信开源理念而创办 OSAF，另外一个原因是在从风险投资事业中抽身后，他意识到设计软件才是自己的事业。但当 Chandler 的日程计划在软件时间中越陷越深，完成日期就像海市蜃楼般不断延后，局

[①] 名字源自林肯在葛底斯堡演说中的著名句子：...and that government of the people, by the people and for the people shall not perish from the earth.（民有、民治、民享的政府将永存于世）。

内人和局外人都在猜，为了项目的进展，他还会有多么投入。

在 2003 年 10 月对外宣告时，卡普尔自掏腰包 500 万美元来支持 OSAF。2003 年，基金会从梅隆基金会（Melon Foundation）和多个大学科技院系组成的联合会——共同解决方案组（Common Solutions Group）得到追加支持，总计 275 万美金。随着人员花名册不断拉长（2004 年 5 月是 24 人，计划增加到 30 人），新募集的钱虽然可以支持更久，但也无法永远支持下去。

现金是软件工业的终极约束，它将项目从软件时间的虫洞拉回到现实中。卡普尔有的是钱；如果 OSAF 需要，谁都认为他会给——只要他还热心于 Chandler。技术学者艾斯特·戴森（Esther Dyson）曾笑言卡普尔在 20 世纪 90 年代早期成立的公司 On Technology 是“生而口含银匙”。OSAF 嘴里的汤匙可能只是镀银的，但对于开源项目，任何形式的继承都不寻常。

当人们直言问及卡普尔对于 OSAF 的态度，他总是回答说：“这是我毕生的事业。”他拿出自己和妻子刚共同签署的长期租约，指出他们合租了沃德街办公楼的另一层办公室，让 OSAF 可以扩充规模，也留下空间给那些他们打算置于羽下的其他组织。

但毫无疑问他撞了墙。工作中毫无乐趣可言：“就像是水在一夜之间冻成了冰，”他后来告诉团队成员们。“是一次相变。”他不会退出，但他总得做点什么。他和妻子做了多次探讨。她有了个点子：为什么不问问 OSAF 的其他经理，让他们重新设计你的工作职责？

5 月 18 日，就在每周四的经理例会开始前，卡普尔发出一封主题为“额外事务”的邮件：

我热爱 OSAF。我坚信它未来一定会成功。我们刚租下 4 楼 10000 平方英尺的办公室。

咱们正处于项目的拐点。我能感觉到。现在有那么多富有生产力的新人加入团队，大家都集中精力打造一个可运行的产品，项目取得如此大的进展，我统管一切的老法子完全不适用了。有必要出台新的管理方法。

……是时候重新设定我的工作职责了。我需要你们帮忙。我出过很多力，但显然对于有些管理工作而言我不是最佳人选。我妨碍了、或者说没有带来最好的技能组合。我想做正确的事。我需要帮助。我承受得了改变。

卡普尔要求团队自己开会，他不出席，会议结束后给他一份建议。紧张的几个小时之后，他们有了计划：米奇应该多做全职领导人，少做实际管理工作。他应当“授权”给经理们，给他们更多自行决策权。没必要到外面专门聘请一位开发经理，不过要做些改变，改善组织机构。帕兰特已经有了个新职务，做大的工程决策的“架构协调人”；应该让丽萨·杜索特负责细节计划和开发工作进度的制定。

卡普尔很高兴。“咱们达成一致了，”当他的同事们告诉他这些主意时，他这么说。一个星期后，他坐在最喜欢的位置、给自己不舒服的背部垫了个小靠垫、PowerBook 电脑在他的灰白头发和胡子上映出冷酷的蓝色，他向全体人员宣布了这个计划。

“应该期望有组织的机构保持周期性演化，”他告诉他们，眼光扫过屋子。“最大的挑战是，咱们不能让野心泛滥，但也要防止灵感丧失。有个问题时常让我失眠：咱们做得怎么样？咱们想要尽快做事，又不致用尽人力。怎么才能快速前进又不损丁折将？怎么才能在项目周围建造一个社区而且不影响设计的完整性？怎么才能从良性独裁过渡到小组民主？”

“为了实施要做的事，咱们比以前更加组织结构化了。和以前的透明制度不同，这更像是个常规的软件开发项目——我想，必然如此。怎么才能理解这只是个形式问题？随着时间推移，咱们不是在挖洞，而真是在

做一个从参与者自身的角度看来有更大参与度和自主性的项目。把这事摆上桌面很有用，在做下一步干什么的重大决定时，咱们都会想起这个。”



“米奇声名卓著，”在我问到为什么 Chandler 项目花费了这么长时间的时候，安迪·赫兹菲尔德告诉我说：“他不想被扔鸡蛋。所以我们有点太过谨慎，像是一个倾尽所有只得代码的 17 岁少年。我们倾向于非常、非常地小心。”

不过，卡普尔难道还没有证明自己吗？我问道。艾瑞克·瑞蒙德说过，开源程序员们的驱动力来自于在同道者中获得荣誉的饥渴。卡普尔早已多次成名。对他来说承担风险不是很容易吗？

“不，他还需要更多荣誉，”赫兹菲尔德答道，“作为设计师，我们都需要更多荣誉——来展示自己能设计了不起的东西。初次成功的人，特别是年轻时就取得成功的人，究竟是靠运气，还是靠本事？嗯，两者都有一点。如果你能做到另一件了不起的事情，就能让世界看到你的实力。”



每当看客们对 Chandler 的缓慢进展表示怀疑时，卡普尔总会抬起他严肃的眉毛，评论说：“如果不坚持，我什么都不是。”毕竟，他不停地听到有人想要把 Microsoft Outlook 和 Exchange 程序扔到阴沟里去，代之以更好、更灵活和更稳定的软件。就在经理们重新设计他工作职责那个星期，阵亡将士纪念日那个周末前夕，他妻子的非营利机构推平游戏场研究院（Level Playing Field Institute）的 Microsoft Exchange 服务器遭遇了悲惨的崩溃。LPFI 工作人员损失了三天的电子邮件和日程计划。

感觉 Chanler 永远也做不出来了。但是坚持，卡普尔坚信，总会有回报。

第 8 章　白板上的即时贴

[2004 年 6 月～10 月]

在微软和许多态度严谨的软件公司中，长久以来都有一种规定，即“吃你自己的狗食”。别误会，比尔·盖茨可没侮辱他的员工；这种说法的意思是开发者必须使用自己正在做的产品。（该说法可以追溯到过去的电视时代，当时罗恩·格林[Lorne Greene][①]宣称爱宝牌[Alpo]狗食非常好，他甚至用来喂自己养的狗，以此作为推销手段。）

一位微软开发经理在 blog 上这样解释道：“吃狗食……是我们内部主动采取的一种手段，用来在服务器发布版中找出最后一些产品缺陷。我们发现，在测试版软件上运行薪资、电子邮件、源码控制和健康福利系统，是在开发周期中尽早提升产品质量的有效手段。;-)”

在 OSAF，犬只四处游荡，旗舰产品上也描绘着一张狗脸，“吃狗食”感觉很对路；正如米奇·卡普尔所言，它有一种“积极的内涵”。2004 年上半年，Chandler 的开发工作蹒跚前行，卡普尔和他的团队开始越来越坚信，获得更好进展的关键是将软件改进到他们自己可以使用的程

[①]加拿大演员，后到美国发展，主要作品有《星际争霸战（Battlestar Galactica）》等。20 世纪 70 年代初，他为 Alpo 做品牌代言，拍了一系列商业广告。

度——让 Chandler 成为可以吃的狗食。



图 8-1　罗恩·格林为爱宝牌狗食做宣传

“吃狗食”的说法可能更多地和微软联系在一起，但更早以前就有人实践过。在传奇般的施乐帕罗·阿尔托研究中心（20 世纪 70 年代发明了现代个人计算技术），研究队伍领导人鲍勃·泰勒（Bob Taylor）奉“吾做吾用”若圭臬。他的说法只对道格·恩格巴特的“提靴带”理念做了些简单修改。“提靴带的目的是推动反馈循环，今天用昨天发明的工具为明天打造更好的工具；而吃狗食则是迫使开发者把鼻子伸到产品的问题中、加速发现和修正缺陷的低调且实用的方法。”

卡普尔心目中总有一位吃狗食的人选：他自己。他想要一个日历软件，好让他和助手艾斯特·孙（Esther Sun）都能修订日程并且与对方的日程保持同步。Chandler“读-写式 P2P 共享”就是为支持这类需求

而设计的。他们尝试了苹果公司的 iCal 软件；iCal 对于只读式日历信息共享——本质上就是向大批受众发布信息——相当不错，但就是不足以让两个人之间做“读-写”共享授权。

如果要把 Chandler 做成可吃的狗食，就该提供一种卡普尔和孙能共享信息的日历。也就是实现 Chandler 关于提供直观、万能的信息共享的承诺。然而共享功能仍然如同一年以前 Chandler 团队所形容的那样，是一条拦路大蛇。设计组还没有解决与之相关的无数开放性问题，程序员们也都还不知道该如何去实现它。

路上毫无指引——没有可供参照的成功系统。卡普尔离开莲花公司后，雷·奥兹（Ray Ozzie）[①]开发的 Lotus Notes 算是一种方案，但它太依赖于重量级的服务器，而 OSAF 则要实现一种无服务器、P2P 的方案。离开莲花公司后，奥兹开始打造一种名为 Groove 的新共享协作系统（卡普尔是早期投资者之一）；Groove 是真正的 P2P 系统，解决了该种设计的许多问题——包括保持用户数据同步、维护信息安全（在后 9·11 时代 Groove 日渐增长的军方客户最重要的关注点）等等。但 Groove 只能在 Windows 上运行，而 OSAF 则承诺让 Mac 和 Linux 用户也能共享信息。

越是深入考虑共享问题，OSAF 的每个人就越怀疑是否存在太多互相矛盾的需求。从来没有人做过不依赖服务器就能在不同计算平台间无缝共享信息的软件，或许自有其原因在，或许根本不可行，或许 OSAF 需要重新审视自己的计划。

1010101010101010110101101110

这就是丽萨·杜索特的看法。杜索特于 2004 年 3 月加入 OSAF，负责和凯蒂·帕兰特一起管理开发者。在几周内，她们将扩张到过于笨重

[①] 奥兹创办的 Groove 公司后被微软收购，奥兹本人也成为微软首席技术官。


丽萨·杜索特

的应用组一分为二：经过改造的应用组继续建造 Chandler 前端 GUI，而新成立的服务组（Services team）则集中于在资料库之上但在 GUI 之下的层——类似邮件传输和共享机制等功能——这个层在幕后与其他软件交互。

杜索特是一位聪颖、热情、做事井井有条的加拿大年轻人，说话措词严谨，爱好空手道和编织，对互联网标准的奥义饱含激情。早在 90 年代初她还读大学时就到微软公司实习，1996 年毕业后加入微软，任程序经理。当时微软正忙于撕开 Windows 的表皮，往其软件核心植入互联网访问和网页浏览器特性。（之后这成为了美国司法部与微软之间反托拉斯战争的核心问题。）杜索特在数个不同的服务器项目中工作过，包括在许多个周末折磨过米奇·卡普尔的 Microsoft Exchange Server 的早期版本在内。

杜索特的到来创了 OSAF 好几项记录。她是第一位加入项目的微软高手；受新老板（“他是位退伍军人，一向当说才说”）的利诱，她于 2000 年离开那个软件巨人，在一家初创公司中负责开发管理数年之久。她也是 Chandler 团队中第一位曾为互联网标准世界（Internet Standards World），一个负责监管核准网络工程标准——用于打造兼容软件的被广泛接受的、开放、良好文档化的协议——的松散机构工作的成员。在微软工作之余，杜索特就开始为互联网工程任务组（Internet Engineering Task Force，IETF）下面的标准委员会工作，并参加 IETF 的各类会议。

在 IETF，杜索特参与制订新标准 WebDAV（Web-based Distributed Authoring and Versioning，基于 Web 的分发著作授权及版本协议）的工作，WebDAV 是 Web 基础协议的一个扩展。其官方站点这样解释：“（该协议）允许用户协同编辑和管理位于远端 Web 服务器上的文件。”直到 2002 年，她受命主管 WebDAV 工作组，还写了一本关于这个主题的书。

对于许多程序员来说，标准领域是一片令人畏惧的沼泽地，充满没

完没了的草案修订、邮件列表上的掰手腕游戏、以及将普通的软件开发搞得看起来像是举办 Indy 500 车赛[①]一般令人发狂的超慢进度。(大软件公司因常在草案尚未通过时就实现新标准而臭名昭著。)但杜索特却无视其进展之缓慢，反而灌注了极大的热情。

"我想，这就是我看起来比大多数人花更多时间在上面的原因——我看到了生命周期，"她说，"多数人以一年为时间框架。软件开发者这样就好。写出你根本没想过要用 5 年的代码？很难知道怎么才能做得对。一年可能都太长。所以，用这样的眼光去看标准制订工作，看到工作组至少要花两年时间做草案审订，再花六个月时间通过草案——这还没完，只是完成了框架性工作，还得做剩下的部分——当然会觉得花的时间长得可笑。你要用以 5 年为计量标准的眼光看问题。得花上 5 年时间，才能得到你真正想要的有用之物。"

这并没有让杜索特感到困扰。"在互联网标准世界，我认识了一些胸怀远大的技术人员。他们并不考虑明年夏天发布的版本，或者下周要修正的缺陷。他们只思考怎么修改类似电子邮件这样已经存在 15 年的标准，或者发明他们期望能存在 25 年的新标准。"

杜索特说，要做到这一点，得像城市规划师一样思考。"以电子邮件系统为例：它是一种容纳了软件、真实服务器、用户账号及其他所有一切的不断变化的组合，拥有非同寻常的稳定性。它是几种几乎非由任何人设计的自然系统之一。人们设计其中的环节，安装上去，它就开始自然变化了。所以，我通常的做法是，当你把系统的一块新部件放进去，总要先看看之后 5 年或 10 年间自己会不会后悔——你是否能扩展它、替代它。"

到 OSAF 后不久，杜索特就利用午餐时间向同事们做了个关于 WebDAV 的讲座。她将它描述为一种"秘密协议"：内建到 Microsoft Windows（作为网页文件夹）和苹果的 Mac OS X 系统中，也在许多需

[①] Indy 500，每年在美国印第安纳州印第安纳波利斯举办的汽车赛，赛程长达 500 英里。

要提供某种文件远端共享和编辑功能的软件包中存在，但却鲜为人知。“人们顺手捡起 WebDAV，开始使用它，因为它看起来不错。正合用。无须太多修改，就能为你所用。”

WebDAV 的工作机制是扩展 HTTP——Web 服务器和浏览器之间赖以互相通讯的协议——增加了让用户在远端服务器上编辑文件的新命令。WebDAV 的创建者们解决了 Chandler 开发者在实现共享功能时遇到的一些棘手问题。例如，WebDAV 内建有冲突消解的机制——当两个不同用户试图同时修改一份文档时，决定怎么应对。或者通过“上锁”避免冲突发生，即确保在用户编辑文档时，其他人不能访问该文档，直至编辑结束、修改得到保存。

如果 WebDAV 能做到，为什么 Chandler 很难做到呢？Chandler 采用 P2P 方案，意味着不会有中心服务器作为开发者不无敬意地形容的“真实源泉”。WebDAV 服务器保存文档，知道用户对文档的操作，也能在多个用户间协调传递关于文档状态的信息。采用无中心的、P2P 的方案，就会产生一份文档的多个副本，没有用户可依赖的原稿，导致无据可查。

如果没有“真实源泉”，日子就更难过了。对于程序员和其他人来说，拥有标准权威会比较方便——可以将你从自己判断两难问题的困境中解救出来。在 OSAF 工作了几周后，杜索特确信 Chandler 用 P2P 实现共享极有可能是死路一条。无论如何，项目至今仍没太多可以示人的。“根本就没有可运行的 P2P 代码，”她后来告诉我，“但看似我们在做 P2P 软件。我们必须这样做。我们说过要这样做；我们决定这样做。”

程序员们总会带着既有的热情和专业知识去解决新问题。从坏处想，这将导致“手上有锤子，到处是钉子”的错误匹配局面；从好处想，这意味着当有新人加入时，项目就能搭一段他们的技能快马。不管是哪种情况，从杜索特迈入 OSAF 大门的那一刻起，看起来 Chandler 最终以某种方式采用 WebDAV 会是个不错的赌注。实际上，加入项目后，杜索特

开始安静地为 Chandler 的“共享传输”制订新方案——采用 WebDAV 的方案。

新方案会用上服务器——这可是 OSAF 的一个大逆转。卡普尔花了时间才接受这个主意。对于 Chandler 程序的 Westwood 版本，就是那个为大学校园定制的版本，总会有一台服务器在某处存在。但卡普尔明白，在 Chandler 第一版，即他们所说的 Canoga 版中采用有服务器的方案，无异于一种妥协，甚至是一种屈服。局外人可能会感到失望；OSAF 开发者们可能会抵制该方案。但若 Chandler 还想继续支持共享——也必须支持——那么 WebDAV 就有压倒性的优势：它已经做好了共享功能。

不管怎样，卡普尔开始重新考虑 P2P 的优缺点。在 6 月份的一次会议上，他向团队正式宣布采用基于服务器的共享设计，他解释道：“我的观点有了重大变化。原来我太过前卫和理想化，立意虽好，奈何不实用。重要的是授人以能，而非架构本身。也许我们需要一种强固的服务器公共架构，好让人们有所作为。OSAF 和我的出发点是，电力是好东西——所以每个人都该有自己的发电厂！我总是想当然以为所谓授人以能就是采用无服务器或轻服务器方案。现在我觉得那种想法大错特错。”

几个星期后，他阐述说：“这简直成了个感情问题。所以，那些推崇自由与创造的开源思想领导者们，将这种观念牢牢绑在自负的美国神话上——该干嘛干嘛去。关于这个话题可谈的很多，但开源自身和互联网要比狭隘的 P2P 自由主义准则来得更具协同性。”

至少有一部分 Chandler 开发者没把 P2P 当作一种意识形态或一种信仰，但却是他们确定的主要愿景，而且还写了一些支持 P2P 的代码。所以向开发组兜售新方案要花点心思。资料库开发者安蒂·维达尤其抵制新方案。

“我们得选一条前进之路，”在 6 月份的会议上，凯蒂·帕兰特解释

说。“我们无法一步到位。”杜索特说。

维达反对说，运行服务器意味着要求用户将自己的数据拱手交予他人。“我不用.Mac，也不用 Gmail。我不信任第三方持有我的数据。无论如何，对我来说，这不再像是共享——更像是向外公布这些数据。”

在搭建 Chandler 资料库时，维达着眼于支持资料库之间的同步和共享。他认为，这是项目的基本要求。卡普尔总是说，希望 Chandler 让用户便于在工作机和家庭机之间保持数据同步。

“米奇扔掉了使用 Chandler 的一条理由。作为最终用户，如果你不采用 P2P 方案，”那天晚些时候维达告诉我，“实在令人失望。而且情况会越来越糟，因为这类事情会越来越多——我们会说，喔，这东西太难，最好放弃吧。基本上他们没错，宁肯发布没做完但是能用的东西，也不要为全部完成再等上十年。想得挺好。不过也令人痛苦和丧气。用加上服务器的新方案，Chandler 只会变大。人家可没想到这个，光想着，那样会更容易做，比资料库共享更简单……也许我错了。知道错误原因，犯错也值得——很不错！可以学到点东西。不过我有点担心，因为项目越来越大，而不是越来越小。”



采用基于 WebDAV 的新计划，当 Chandler 用户想要共享一组条目，如一份日历时，Chandler 就会将该组条目复制到一台 WebDAV 服务器上。共享者将登录到该服务器访问日历——读取或修改。服务器确保所有修改都会回传给原用户。还有一些尚未解决的问题：如何让你知道我想和你共享？服务器如何控制对不同用户的数据存取？如何终止共享？不过最大的问题是“Chandlerness[①]”。

Chandler 条目不是简单的文本块——而是附着了一组属性的软件对

[①]即维持条目作为 Chandler 条目的本质。

象，包括是什么、属于什么群集、如何互相关联，等等。WebDAV 的设计目的是协调网页和类似文本文档的协同。在发送到 WebDAV 服务器时，Chandler 条目会发生什么变化？它们还会是 Chandler 条目吗？需要做什么额外工作？

回答这些问题的唯一途径是开始搞代码。当时，丽萨·杜索特已经在做一个叫做 CalDAV 的项目，主要用于扩展 WebDAV、使其在处理日历方面更加灵巧。假使 Chandler 采用 CalDAV，并且其他日历软件也支持 CalDAV 标准，则 Chandler 用户就能与其他日历软件用户之间实现共享了。最终，杜索特和帕兰特期望能创建 WebDAV 扩展的一种更进步版本——称作 ChanDAV——它可以让 WebDAV 处理 Chandler 生成的各种动态变化数据。

目前，任务稍微简单些：设置一台 WebDAV 服务器，让 OSAF 开发者们使用，编写将 Candler 条目发送回服务器的代码。其余之事，包括 CalDAV、ChanDAV 及其他尚未命名的 DAV 版本，仍然远在他方。

但就实现共享功能的现实计划而言，仍有拦路毒蛇嘶嘶作响。



2004 年 6 月，又是一个土拨鼠日的时节：OSAF 的经理们和程序员们从显示器后面伸出头来，环顾四周，意识到自 Chandler 0.3 版发布已有四个月之久，按照时间表，他们不可能在三到四个月之内发布 0.4 版了。

月末的一个星期一下午，林智超和丽萨·杜索特走进米奇·卡普尔洒满阳光的办公室，在靠近门口的小圆桌旁坐下。卡普尔也参加了讨论。议题：制订 Chandler 1.0 最终版（Canoga）的开发计划，该版本看起来仍然像是高不可攀的顶峰。

“我们采用特性驱动还是进度驱动？”杜索特问。

“应该两者都有点儿，”林说。

“在真实世界中，总是两者皆有，”卡普尔表示同意，“独沽一味不正常。咱们的计划是特性优先。当然还要不断剪裁和调整。就像是个双星系统，一颗星星比另一颗大些——彼此影响运行轨道。”

“现在我相信吃狗食原则，宁可快一些得到可用的东西。反对者认为，在某个特定时间点发布功能不全的版本是一种冒险。但常言有云：‘半个面包总比没有强’，足以驳倒这种观点。”

他们开始复查林填写的那张包含 **Canoga** 可能特性的 **Excel** 表。开发者们还有大量功能没有开始做——类似数据的导入导出，或打印，或“版本控制”（跟踪用户做的修改，以便能够取消操作）。在他们探讨每个额外的话题时，能量似乎就要爆棚而出。

“我刚加入时就认为共享是座冰山，因为我真的很熟悉这个领域，”杜索特说，“其他领域我不太熟，不知道在水面之下是否也有冰山。”

“就不能做一份没有冰山的好计划吗？”卡普尔问道。

杜索特回答说，即便如此，实现 **Chandler** 的所有特性仍需假以时日：加快进度的手段有限，但如果聘用太多新程序员，布鲁克斯法则就会起作用。

“没错，”卡普尔表示同意，“就是那句‘十月怀胎，欲速不达’的老话。”

“我想从现在起一、两年后才能发布 **Canoga**，”杜索特说，“至少。”

卡普尔长叹一声：“何以见得？”

“要做的事多到令人恐怖。据我所知，难倒是不难，不过太费工夫。我对照了 **Exchange Server** 发布两年前的功能模糊状态，还有他们砍掉了

多少特性。”

“而且还只是 Canoga 版，”卡普尔说。他轻叹了一口气，像是吹了个长而无声的口哨，闭上眼，双手盖到嘴上。沉寂良久。然后，他有些敷衍地反驳杜索特：“很难对照软件的将完成率。新人刚开始解决电子邮件和 WebDAV 的问题。自从 0.4 版本后，工作的性质也在变……”

“可是我看到了凯蒂小组未来四个月的计划，”杜索特答道，“包括所有的选择对话框和打印对话框——得花时间，得慢慢做。”

卡普尔认输了，继续下一个话题。“下面的问题是，什么时候能拿到可用的狗食版本？”

“狗食版应从小处开始，由几个专注于此的人来做。”

“如果真的把关注的焦点放在狗食版上，那扎扎实实干上六个月还差不多。肯定得做好些痛苦的决定。”

“0.4 版发布后六个月？”杜索特问道，面露怀疑之色。

“对。从 9 月份为 CSG 会议（那个大学机构）做好那个版本后开始算，明年 5 月如何？”

杜索特看向窗外，目光穿过沃德街。“如果人们对数字要求过高，工程师通常就会胡乱捏造。”

在类似这样的会议上，我一般都作壁上观，默默记录。不过这次卡普尔转头过来，跟我开了个玩笑，“司各特会给这章题名为‘面对现实’。”

我点头微笑，心想，**软件时间又来了。**



11:34 hazmat[①]> 你知道，chandler 组件/包的示意图会对外面那些程序员很有用，同时有益于他们的不稳定状态

[①]这是一份在线聊天记录，所以人名和软件名称等都不是大写的，句尾也没有标点符号。

11:34 katie> hazmat：对，我们大概可以在下周安排做这类事……
11:36 hazmat> 太好了，修动太大，很难知道有些什么变动
11:36 katie> hazmat：哥们儿，什么都在变！
——来自 2004 年 7 月 14 日的 Chandler IRC 聊天频道

有一帮子外面的人定期登录 IRC 频道，跟踪项目进展，询问核心程序员们一些问题。Chandler 程序员偶尔也会收到外部程序员发来的补丁包——修改一些问题的代码块。Chandler 采用开源模式，即代码公开，但 OSAF 还没能推进项目的集市式志愿者贡献。“别人对我们特别有耐心，”泰德·梁在 blog 上写道，“我希望在软件做到半可用状态时，他们的耐心能转化为积极的支持和贡献。”

内部程序员继续做基础工作。戴维·萨鲁维全力投入，为应用组需要的 wxWidgets 组件做修正和改进工作。他甚至在解决约翰·安德森早在 0.1 版时就提出的“闪烁”问题上取得了一些进展；看来在 Chandler 运行的三种操作系统上情况各有不同，这与操作系统处理屏幕图像的“双缓冲”机制有关——在屏幕之外准备窗口的可视内容，当用户点击时就能马上显示。

同时，唐·登曼也做好了细节视图和打戳功能。电子邮件专家布莱恩·科斯齐将基于 Python 的 Twisted 开源框架集成到 Chandler 中；Twisted 能处理多种不同类型的网络通信，负责 Chandler 收发电子邮件的许多幕后操作。约翰·安德森忙于“重构”CPIA 构建块代码，使之更加紧凑高效。

“比如，”他有一次在团队会议上展示成果，“原来处理资料库视图的代码是 121 行，现在成 85 行了。”

“代码行数的度量是个巧手活，”卡普尔指出，“我们刚减少了 36 行代码。”

有两位新加入的成员，负责让 OSAF 的开发进程向更为结构化和规范化的方向发展。7 月，曾在网景和硅谷其他公司干过的老手阿帕那·卡达基亚（Aparna Kadakia）成为 OSAF 的 QA（质量保证）经理，开始为 Chandler 建立完整的测试程序。8 月，雪拉·莫内（Sheila Mooney）——最初在微软做多媒体工作，后来在一家医药软件公司从程序员升职为经理——参加了林智超的产品管理工作，为 Chandler 的各个方面编写更为正式及细致的规格说明文档。

为 0.4 版设计的主要特性还有很多未完成，而时间已经是 8 月份了。史都沃·帕门特负责在 0.4 版中用 WebDAV 实现八字还没一撇的共享功能，他还有很长的路要走。约翰·安德森本来进展良好，不过随即在他所谓的“老鼠洞”问题上耗费了许多个星期。时钟在嘀嗒作响：自从 0.3 版发布以来，已经过了 6 个月。0.4 版的计划要求在 9 月初固定特性，然后进入缺陷修正和文档编写阶段，在 10 月底全面完成。卡普尔、帕兰特和杜索特都知道，他们将错失狗食版本的目标。

有好几个月，他们跟踪每位开发者的任务和计划，要求每两周发布一次里程碑版本代码。这有助于推进 Chandler 项目吗？很难说。不过他们知道，这耗去了自己大量时间。

“‘未完成’的鸿沟仍然难以跨越，”卡普尔在 8 月 10 日的复查会议上说，当时经理们正在浏览长长的里程碑目标最新列表，包括完成的和没完成的，“咱们的目标是更准确地预测。”

“我先唱个反调吧，”帕兰特说，“总是按计划执行远没有保持较快进度并清楚做到哪一步来得重要。如果老往计划里掺水，从而达到目标，就会变成退而求其次，完成的特性更少。”

“人们不善于估计完成度，”杜索特建议说，“而对剩下多少时间更有把握。”

卡普尔谨慎地说："我想强调一下，跟踪进度是为了协调工作，而不是要表扬或者批评谁。"

"目前的进展对史都沃完成工作毫无帮助，"杜索特说，"改变不了未完成状态。"

"有个大问题，"帕兰特说，"就是工程师说'完成啦！'阿帕那及设计组查看后说，'你说什么？'换言之，特性或许可以'运行'，但还有一大堆缺陷，而且界面一点都不像设计的那样，后面要做的事还很多。"

泰德·梁通过会议室里的电话会议装置加入到讨论中，他说："写下这些列表的要点是，我们尽力操纵项目进展。没信息就做不了这个。问题是，我们需要多少信息？"

"我在尽力保持耐心，"卡普尔说，"不做蠢事。即便接受有东西没做完的事实，也需要了解做了些什么。否则，就等于是引火自焚。"

会议的惯例是逐组汇报——应用组、服务组、资料库组。在别组说完自己的麻烦事后，梁汇报了资料库的开发工作进展情况——有两个小项目在按计划进行，没有什么意外情况发生——令人感到安慰。卡普尔做了笔记。

"很有意思，之前我们在资料库上投入了无数时间，每个人都牵扯进来，有时一次苦耗上几个月。现在，我们要做一些以前没做过的事，又撞到墙上了。不过项目还是在往上攀。我想记住这个会比较有帮助。"

"对我来说，作为一名软件设计师，如果你设想一种特性，随后这种特性就魔术般地出现，那真是太好了。不过现在我们身处地狱般的境地。有那么多好主意要拿出来毙掉。一茬一茬地。"

"好消息是，上一次在 On Technology 遇到这种情况时，在走得太远前我就叫停了。这回，我要做到底。"



与此同时，距此地 800 英里以北，类似的一幕软件时间大戏也正上演。微软自 2003 年夏天以来就大肆宣扬新的 Longhorn 操作系统，不过——大吃一惊！——2004 年微软放慢了 Longhorn 的进度。夏天，微软宣布新的 Windows 操作系统不会在 2006 年底前发布，而且也不会包括许多在一年前曾经自豪鼓吹的特性——包括那个美妙的、听起来与 Chandler 无地窖式数据处理手段相似的新 WinFS 文件系统在内。

但当 Longhorn 和 Chandler 都步履维艰时，在 Web 世界里却发生着翻江倒海般的大变化：新一代软件产品以基于网页浏览器界面的形式出现了，而且，奇客狂人们宣告说，和它们的前辈不同，新一代软件**很像样**。4 月 1 日，大获成功的搜索引擎公司 Google 发布了一种基于 Web 的新电子邮件服务，这种新服务叫做 Gmail。这举动出人意料，在愚人节发布的举动也让许多观察家将其视为恶作剧。但它千真万确。虽然关于 Gmail 的喧闹之声都集中于 Google 提供了前所未有的容量高达 1GB 的免费邮箱，但这项服务真正的创新之处却在于“堆到一处”的信息管理方式。你不用把邮件分类到文件夹里面，所有邮件都堆在一起；Gmail 能智能化地按对话线索排列邮件，并帮助你使用 Google 的独门搜索工具查找东西。

Google 也以类似 Chandler 的方式，撬开了至少一组信息地窖。它也提供了一种简约的界面，通过某种高超的编程技术，能比其他基于 Web 的程序更快地响应用户点击。Gmail 消除了浏览器在用户每次操作时与服务器进行信息交换的需要。（这个问题也是米奇·卡普尔不愿把 Chandler 做成基于 Web 的应用程序的原因之一。）它使得网页浏览器如同 Chandler 将要成为的那种“富客户端”应用程序一样工作。它减少了

桌面应用和 Web 应用之间的不同。

Gmail 并非炫耀这种魔术——后来被简称做 AJAX（asynchronous JavaScript and XML，异步 JavaScript 和 XML）——的唯一产品；有些小公司也在玩类似的酷把戏。位于英属哥伦比亚温哥华（Vancouver）的一家小公司开发出 Flickr，这种基于 Web 的数码相片共享服务也实现了同样的速度和愉悦体验。还有芝加哥一家小设计公司 37Signals 开发的 Basecamp，在基于 Web 的小组项目管理工具中提供了类似特性。

这类产品没有立即占领大众市场，但赢得了一些热情的“早期接受者”——奇客和技术作家，他们在自己的 blog 里大肆宣扬，让人感到新一代软件正在崛起。卡普尔看得比较长远——这在浮躁的技术界是一种健康的做事方式——但他知道 Chandler 并不在真空中成长，它的成功取决于是否能实现做出明显更好的信息组织方式的承诺。

我和几位对 OSAF 友好的观察者谈过，他们坦承了自己的忧虑：如果 Chandler 还不加快速度，他们担心，就会赶不上潮流。

1010101010101010110101101110

Chandler 团队采用“kibble（粗磨）”而不是 Canoga 作为第一个狗食版本的非正式名称，他们自己将每天使用这个版本的软件。里面到底会有什么？需要多长时间才能做出来？怎么做到呢？

9 月 7 日，凯蒂·帕兰特、丽萨·杜索特、林智超和雪拉·莫内在一天的长会结束前，聚到一起开了个“kibble 计划会”。

卡普尔和林将在两周内与 OSAF 的资助者、大学联盟 CSG 会面。他们想要一份清晰的计划。林给同事们发了一份巨大的电子表格，列出 Chandler 的全部特性，再分解为驱动他们工作的单个任务。

这份文档吓死人。他们四个围坐在大会议桌旁，盯着眼前笔记本电脑屏幕上的表格。

“过一遍要多少时间？”杜索特问，“会不会要整整两个星期啊？”

“有别的办法吗？”帕兰特回答说。

杜索特盯着表格上的行和列。“我们得想办法了解一个发布版本到底有多大。将特性放到同样大小的篮子里。我们没法用这种格式做对比。我们只知道在以前版本中做了什么。”

“可以这样做，”她建议说，“用贴纸，每张纸表示大致同等的工作量。”每张即时贴代表单个开发者一个月或两个月的工作时间。先在墙上循“点号版本”的顺序贴上，然后就能对每一轮计划的工作和自己是否脱离现实一目了然。

“白板上的即时贴？”雪拉·莫内说，“我以前用过。我之前那份工就是这么干的。”

有那么一阵子，似乎他们打算另开一次会讨论——已经过了下午 5 点钟。但是杜索特跑出去，带回一堆即时贴。她擦干净屋前那块长白板的左边三分之一，在上面画出四栏，标上 0.4、0.5、0.6 和 0.7。然后，她又画了两条水平线，将四个竖栏划成上、中、下三个部分，对应 OSAF 的三个开发组——应用组、服务组和资料库组。

杜索特、帕兰特和莫内开始写贴纸，一张张贴到白板上。一边贴，一边讨论任务名称、粘贴位置、是否把某些特别困难的任务分到两张贴纸上等等。“核心版本控制会不会太大？”“搜索占一张纸好还是两张纸好？”

白板上贴满了一组组用黑色马克笔写的黄色贴纸。数量少一些的蓝色贴纸代表后端任务——不涉及用户界面的编程工作。还有很少的紫色贴纸，代表与项目架构有关的工作，例如源代码控制系统或者构建系统。

在十五分钟内，他们就贴完了 50 张贴纸。半个钟头过后，贴纸数量直逼 100 张。

莫内转过身来，咧开嘴笑了："此计大妙！"

几位经理依据林的列表工作，略去了已经被取消或推后的特性，还有在狗食版中没必要包括的特性：类似尹咪咪提出的"鱼眼视图"，这种视图将信息的展示集中到最相关的条目或"条目对话"——用户可以附加到共享条目后面的即时信息式标注。每个取消掉的特性都至少占一张贴纸，也许更多贴纸。但最后还是有 138 张贴纸：0.4 版的 27 张、0.5 版的 45 张、0.6 版的 33 张、0.7 版及以后版本的 33 张。

他们从会议桌边拖过椅子，坐到贴纸满幅的白板前，盯着贴纸看。

莫内大声说道："可叹 0.4 版格子"——表示即将完成的那个版本的工作量——"里面东西最少。"同时，0.5 版格子中的贴纸多到溢出。

"而且我们还打算让 0.5 版格子比 0.4 版格子短，"杜索特说，"看来做出 kibble 版的时间至少会 3 倍于 0.4 版。而 0.4 版已经花了咱们 8 个月的时间。"

"我想不能如此类比，"帕兰特反对说，"之前有新人需要磨合。做计划也费时不少。"

"按照我们要做的事来说，我还是觉得做 kibble 版两年时间不够用，"杜索特说。

林以手托腮，说："我想米奇的意思是明年做出来。"

"如果只有两年时间，就要从 Kibble 版里面砍掉一些东西，"杜索特答道，"别忘了，'kibble'版就是名副其实的'粗磨'版。"

"至少 OSAF 自己人能在日常工作中使用，这是基本原则，"林说。

"所以有些东西不必做。例如 PDA 同步"——和移动设备（个人数

字助理）同步。

“那要看你用不用 PDA。”

“以贴纸数量来看，至少要两年。”

“你假设我们不能做得更快。”

“不是每加一个人就能提升生产力的——还记得《人月神话》吧。还有就是要维护的代码越多，要修正的缺陷也越多。”

莫内问：“咱们还要招多少人？”

“我可不愿意看到有人以为招人就能加速 0.5 版的开发进度，”杜索特说，“那种想法不对头。我在想新增人力大概能应付不断增加的维护工作。我发现很难一碗水端平，所以希望两者择其一。”

“如果多招进一倍人呢？”林问。

“第一年会拖慢进度，”杜索特说，“甚至会持续到第二年。第三年才开始有回报。”

“就这么办。”林微笑起来，“好。嗯，有点用。至少有了个数字。”

林和杜索特正在重演一场原型对话——在产品经理和开发经理之间的拔河赛。这种对话在无数的会议室里和无数的软件项目中发生过。它总是如此这般：

产品经理：*时间太长了！不能快点儿吗？*

开发经理：*就得这么久。*

产品经理：*没什么好办法吗？*

开发经理：*如果我们真的很厉害，就能知道需要多长时间。但是我们却无法改变它。*

两年时间做个 kibble 版。如果砍掉一些特性，同时放低眼界，也许

会少花一点时间。

大家同意林将此事告知卡普尔。没人期望他会欣然接受。

莫内写道，他们鱼贯而出，白板上留下一行醒目的大字：**DO NOT ERASE**（不要擦掉）。



星期四，**OSAF** 成员们集合开“全员”会时，那堆贴纸仍然占据着白板的一端。上端是一大堆错综复杂的黄色贴纸，稍下方是占地较窄的蓝色贴纸，点缀着一张紫色贴纸，看起来就像是老电视游戏《太空入侵者（**Space Invaders**）》的屏幕，置于屏幕下端的炮口瞄向一排排目标。一眼就能看见。不过需要解释才能看明白。

会议开始时，卡普尔向白板挥动手臂。“这东西代表了久经考验、最具技术水准、长期使用的工程管理工具。”屋里笑声四起。“它也表示完成度。有助于锁定 **kibble** 版的主要工作。好消息是尚有进度——工作一直在开展。不过显然我们需要严格剔除非本质特性，这样才能在可接受的时间内完成任务。”

接着，林智超回顾了在推动“**kibble** 版更快完成”时可能不幸牺牲的特性列表：联系人管理（通讯簿）。即时通讯。尹咪咪的类 **iTunes** 浏览器。日历里面的月份视图。在 **Chandler** 中将邮件附件作为真实条目的特性。

“这些特性就像我的小狗，”卡普尔说，“这样做好像是把它们束之高阁一般。”

屋里一片寂静。很难说程序员们是放心终于脚踏实地了呢，还是失望砍掉了太多东西。

“公平地说，有些我们真的很想做到的创新看来在第一个版本里面会被剔除。贪多嚼不烂。产品会比我们最初希望的普通得多，更像其前辈。另一方面，还是有许多重要的新特性：概览视图，消除数据地窖，还有共享。咱们仍然要坚持做有趣的东西。只是得尽量缩短战线，尽快做出产品来。”

当月稍后时间，在一次向 OSAF 规模不大的董事会的汇报中，卡普尔回顾了 Chandler 为何迟迟不能面世的一系列原因：他们低估了实现项目的远大抱负所需的代价。组建一个工程团队要比想象中更难。开发“富客户端”软件谁做都难——“不止是我们遇到问题。”还有，从个人立场上看，“我自以为懂得很多。或许我已经变得迟钝了吧。”

没人能指责卡普尔一头扎进领导力的小小快乐考验——出来时干干净净、不沾上失败和过失的痕迹。但他总有办法承认挫败，而又不至于为挫败所打垮。他坦然接受了关于贴纸会议结论的坏消息。他不会向团队粉饰这坏消息——这帮人太聪明了。不过他加了一个现实乐观主义的框子——让大家得以回顾 20 年前是何等优良的品质让小小的莲花公司成功做大。



开发经理们和卡普尔在之后那个星期里一直争论。卡普尔不同意每个后续版本都要和 0.4 版一样花 8 个月时间的假定，开始给出其他可能。一种方案是干脆放弃 Canoga 1.0 版本。开发者全心全意投入 kibble 狗食版，经过 OSAF 人员及胆大的外部人员实际试用之后，再将其打磨为高质量的 Chandler 1.0。然后，开始向用于大学的 Westwood 版本进发。

另一种方案是推迟实现电子邮件功能，集中精力于打造可用的日历功能。林智超在一份题为《不可忍之问题》的备忘录中最早提出这个建

议，招致众怒。用杜索特的话来说，就像是“五脏不全”。帕兰特指出，做出狗食版 Chandler 的多数工作在于使应用程序足够稳定、堪敷使用，无论实现“PIM 功能”（日历、电子邮件、任务、便条）的顺序如何都是如此。

但卡普尔认为，推迟实现 Chandler 的电子邮件功能有助于更快交付可用软件，这是头等大事。于是他们在贴纸板上画了一条横线，隔开做日历功能的任务和其他任务，开始进行所谓“贴纸重构”——撕下贴纸，再重新贴上。

到 2005 年底，在这一年时间里，他们能够完成多少打造可用的 kibble 版的任务呢？

杜索特指着白板上新画那条线的上面部分，即在 0.5、0.6 和 0.7 表头之下、只实现日历特性的部分，说：“今年我们能做的是这些、这些和这些，还有一部分这些特性。”后者位于日历特性隔线以下。

9 月 16 日，在接下来的一次全员会议上，卡普尔给出了新计划：不做 Canoga 版。只做狗食版或半身版。优先做日历特性。“好消息是，到 2005 年底我们就能做出点有意义的东西，我对这个目标越来越乐观。不过还有事要做。不能说已经有可依法炮制的计划。这些不过是我们的假想罢了。”

与会者虽说并未感到震惊，不过也算是被当头泼了一桶冷水。路还长着呢。至少，他们第一次看到有一份脚踏实地、根据对完成特性代码的真实时间记录的估量做出的计划。不再按照期望和断言能做的事“自顶向下”做计划了；现在是根据经验及证据“自底向上”做计划。

在这段风帆调整期中，卡普尔从与 OSAF 毫不相关的一件事中找到了慰藉：Mozilla 项目凭借 Firefox 浏览器日渐成功。Firefox 是从 Netscape 的废墟之上演进而来的、经过改进的 Mozilla 软件现代版本。Firefox 在当年秋天晚些时候发布了 1.0 版本。但即便是预发行版本，也创下一周

内百万次下载的记录，还赢得了大量好评。Firefox 是一种免费的开放源代码软件。而且它是以一种非常类似 Chandler 的项目轨迹发展而来。Mozilla 在奇客精英们的狂热中声势浩大地起步，然后就掉进黑洞达数年之久，失去了一些关键程序员的支持、濒于消亡。Mozilla 一次又一次地错失良机；有了 Firefox，它终于抓住了一根救命稻草。

米歇尔·贝克做过网景公司的律师，在 Mozilla 的黑暗年月里作为管理者和经理帮助 Mozilla 生存下来，她现在又以某种社区顾问的身份为 OSAF 服务。在 Chandler 开发者们尽力消化卡普尔带来的关于进度安排的坏消息时，她说道："在 Mozilla 项目的早期阶段，也有类似的讨论。不过一片混乱。浪费了很多时间，有些是技术问题，但多数要归咎于不能将管理——那会儿还归网景管——和关键的工程推动力结合起来。"

"后来怎么走上正轨的呢？"OSAF 的市场人员皮耶特·哈特索科问。

贝克说，Mozilla 开发者们决定全部重写浏览器的"布局引擎"，在屏幕上画出网页的代码。这一决定的结果是让项目多花了好几年时间，外界对此颇有微词。但是，她说："那是个关键性决定——即便是个错误的决定。然后是设计出可运转的工程进展计划。第三是不断前进、做出基本能运行的东西。"

卡普尔点头说："不觉得听起来挺耳熟么？"

"和我做过的所有软件项目一样，"约翰·安德森说，脸上露出一种认命的微笑。



OSAF 工作环境宜人——午餐很好，工作时间有弹性，气氛非常融洽。但延误和特性削减给团队带来了某种不良的影响。因为受不了缓慢

的进度以及做不出可以为之自傲的项目代码的挫折感，布莱恩·史金勒在夏初离开了。他决定不再回头。秋天，史都沃·帕门特也宣布自己将要离开——加入甲骨文公司资助的一个开源日历项目。大概同一时期，林智超决定调整参与程度；他还会继续在 Chandler 搭把手，但准备重新开创自己的公司。雪拉·莫内适应了项目经理的角色。很快，“鸭子”舍伍德也决定回学校继续攻读计算机科学学位。

如何在项目漫长生命周期的起起落落中鼓舞程序员和他们的经理，是一种神秘的艺术。经历了 OSAF 所有的波折，看来 Chandler 仍有足够的磁性可以吸引住有才华的人们。但类似做日历这种普通的工作却无法叫他们活力充盈。

Chandler 一直具有两面性：一方面，它是用户管理信息的软件应用；另一方面，它是一种“应用框架”或“平台”——开发者可在上面添加新酷特性的一种基础。由于卡普尔和 OSAF 团队逐渐勉强接受了不能两者得兼的事实，他们首先选择做应用程序。“尽快做出可用的软件！”根据以往经验，他们知道这也许是最明智的选择。但他们还是梦想着搭建更大的平台——一种开放的、改变世界的工具，只要添上合适的附加代码，就能管理和连接各种全然相异的数据。

丽萨·杜索特开始将 Chandler 和 CalDAV 挂钩，使其能与更多日历软件互通。唐·登曼将要享受 6 个月的父产假，回来时，他期望能在 Chandler 中创建一种脚本功能，最终可以让非程序员用户自动化 Chandler。安蒂·维达已经完成 OSAF 另一项名为 PyLucene 的开源项目，将 Python 与 Lucene 搜索引擎整合到一起，他梦想着将 Chandler 资料库改成一种更为独立的软件部件，好让其他程序也可以采用。

在登曼展示可工作的打戳代码的那次会议上，摩根·萨奇也向团队做了个演示。他打算为家里人开个相片 blog——有相片的网页日记，但他对现有的软件不甚满意，所以“决定自己用 Python 做一个”，他解释说。

他需要用数据库来存储相片和网页服务器代码。“然后我想到，等等！有什么我知道的项目已经实现了这种功能？”（现在带有网页服务器的 Twisted 网络通信代码已经集成到 Chandler 里面了。）

在同事们全神贯注地凝视中，萨奇展示了他的新相片 blog——满是他小孩相片的页面——这是基于 Chandler 开发的，相片采用 Chandler 应用程序存储和组织，不同之处在于发布到网上，谁都能看到。

这东西本身并无出奇之处。而且仍然粗糙、远未完成。不过对于这群人来说，却是对他们为之奋斗的未来的惊鸿一瞥——而那未来仍是如此令人痛苦地遥远。

有人问道：“做这个花了多长时间？”

“大概两小时吧，”萨奇答道。

第 9 章 方法

还记得质量三角吗——时间、金钱和特性（或质量）？直到 2004 年秋天，也就是 Chandler 公开宣告后两年，OSAF 在其中任何一方面都做得不太好。快吗？不。便宜？花了好几百万美金，而且到现在还没什么能拿得出手的东西。好吗？还得走着瞧。

Chandler 0.4 版于 2004 年 10 月 26 日发布时，一周内大约只有一千人下载，而 Chandler 1.0 在发布 24 小时内就有一万五千人下载。新版本在启动时看来有点更像是真正的软件应用，如果你愿意深入挖掘的话，还能发现一些内建的新功能。第一次出现了部分可工作的细节视图，也可以给条目打戳，使之变为另一种类型。

但对于基础的日常任务——发送或接受电子邮件、保存日历信息、组织便条——Chandler 还是不能让试用者们满意。即便对于它的程序员们来说也不是可以吃的狗食。启动超慢，崩溃频繁。

下载数量的下降不足为奇。世界，至少到目前为止，一直在进步。许多以前为 Chandler 发布高唱赞歌的外部人员抛弃了它，有些正式参与

者认为它迷失了道路。“对我来说，那简直像是一次火车事故，”当我在0.4版发布后不久和安迪·赫兹菲尔德聊天时，他悲哀地摇着头这么说。

然而，OSAF内部第一次充满了宁静的感觉。发布之前的几个星期一点也不狂躁。经理们和QA人员系统检查了Bugzilla中的开放缺陷列表，并且做了筛选：这个本周内修正，那个是微不足道的小问题；这个放到下一版，那个的确是只“拦路虎”，必须得修正。应用组——负责Chandler的外观、用户界面，这部分缺陷最多——有系统地减少着列表上的缺陷数量。设计组在0.4版完成之前就开始计划做0.5版计划，而那些没陷入缺陷修正工作的开发者则开始做与0.5版开发相关的事情。

OSAF也许还没有足可交付给公众的产品。但却另有收获：开始有了工作流程，还有一套可能让它朝目标行进的可行的方法论。

嗯，等一下，你可能会问——难道他们这两年的工作都**无流程可依**吗？不完全是这样。OSAF有很多流程，也许流程太多了，而且这些流程有时看来像Chandler代码一般不固定、常被修改。但在开发0.4版的8个月里发生了一些变化；项目终止了拍脑瓜，建造了某种结构——某种组织机制，可以依靠它向前冲，而不必每几个月就重建一次，或者为每个新版本重新启动其引擎。

从Chandler项目的最早期开始，卡普尔就坚持要做诚实、现实的计划和进度安排。但项目在满足进度方面却拥有不佳的记录：平均6个月能发布一个版本，但计划却总假设应在3、4个月内完成一个版本。部分原因或许是在软件开发和其他领域中计划总是超出了能预见的范围。如丽萨·杜索特所言，“要留心，如果当前计划涉及一年以后，有可能这个计划会失败。”

Chandler和绝大多数其他软件项目一样遭遇了这令人沮丧的现实。软件开发者很少成组地共同开发一系列项目；他们不太像运动队或是军

队单位，也不太像合唱团，而更像是专才们为制作一部电影而临时组合然后解散，再重新为下一部电影组合起来。所以，尽管单个程序员和经理都拥有丰富的经验和技术知识，在过去表现良好，但每次到新团队中开始做新项目时，他们大概还是会按下“重置”按钮，根据某些首要原则设计出一套新的工作流程。

由于成功的流程难以琢磨——甚至有些公司或团队已经定出一套成功流程，也往往会忘记再次利用——软件专业中发明了一整套“方法论”。可以把方法论看作冻干的流程、让软件项目成功的药方。有些程序员信赖方法论——有时择一而从，有时混而用之。而另外许多人则不肯让自己的独特才华屈从于任何条条框框。

自软件行业初期以来，这种或那种方法论的拥趸们就一直在承诺他们的方法论是让软件项目按时、保质、在成本预算之内完成的独门秘笈。这种普适论的问题在于，没有任何一种方法论能够覆盖软件项目的广大领域。适合面向上百万消费者的产品（旧式的塑封包装软件市场）的方法论，未必适用于在公司内部使用的软件开发。在整个政府机关内使用的程序，其对开发者提出的需求异于专用产品，例如音频后期处理。在个人计算机上运行的软件异于在手机或嵌入式电器——在汽车或安保系统内部运行的系统——上运行的软件。最佳实践的药方不可重开。银弹不会再次装入弹夹。



在跟随 Chandler 蹒跚的步履、看着项目伤病缠身的同时，我常常回顾自己在软件时间经验上的反应：**不该总是这样。**有人应该已经想出办法来了。在某地会有一张地图指点人们走出软件时间的焦油坑。

我四处寻找答案，手执灯笼，在软件历史和分析的长廊中游荡，不

断敲门问询："有没有人能放言自己知道做软件的更好途径？"

我开始研究 20 世纪 60 年代，那是个软件创造者们初次系统地审视自己和自己工作领域的年代。1968 年，艾德加·狄杰斯特拉（Edsger Dijkstra）[①]在《美国计算机学会通讯（Communications of the ACM）》上发表了一篇短文，标题有点儿逗乐：《Go To 语句被认为有害（Go To Statement Considered Harmful）》。在许多当年占优势地位的编程语言中，例如 Fortran 和 Basic，计算机逐个运行一系列编号指令语句，程序员可以使用 GOTO 语句在执行顺序中各个点之间跳转。如同电影剪接师使用跳接[②]手法，GOTO 语句也是弃程序一行而不顾，另择一行执行之。它无条件地将控制从程序的一点移转到另一点，对其他因素——不同变量的值或是程序的状态和数据——不管不顾。

"Go to 语句过于原始，"狄杰斯特拉写道，"它是一份请柬，把混乱邀进了程序。"

在 20 世纪 60 年代和 70 年代，许多程序员接受了这份请柬，制造出混乱——枝蔓缠结的软件，后来被称为"意大利面式代码"。"结构化编程"的倡导者们从狄杰斯特拉和其他设想一种更模块化编程风格的理论家那里得到灵感，提出一系列编程实践，试图消除过程式意大利面带来的磨难。结构化编程有几个对立派别；在狄杰斯特拉看来，核心观念是编写由一组子单元组成的程序，每个子单元只有一个进入点和一个退出点。狄杰斯特拉写道，这样的程序不再是一堆面条，而是"一串珍珠项链"，其简洁和明晰的特点不但降低了出错可能性，还让程序员更能理解自己的手工作品。

结构化编程以防御式下蹲的姿态提出建议，尽力让可能犯错的程序员自避其短。"我脑子不好使，头很小，我学会了与之共处，正确看待自己的不足之处，并完全依赖它们。"狄杰斯特拉写道。他在 1972 年获得图灵奖——计算机界的诺贝尔奖，获奖演说的标题是《小程序员（The

[①]艾德加·狄杰斯特拉，1930 年出生于荷兰阿姆斯特丹，2002 年逝世于荷兰纽南。以发现了图论中的最短路径算法（狄杰斯特拉算法）而闻名于世，1972 年因为 ALGOL 第二代编程语言而获得图灵奖。

[②]移去一部分胶片，使得动作产生跳格移动的剪辑手法。



艾德加·狄杰斯特拉

Humble Programmer）》。他的著作中也充斥着对“可怜程序员”的同情，和对人类这种动物“力有不逮”的耶利米哀歌（Lamentations）[①]。

结构化编程的具体规条很快就体现到新一代编程语言的设计和语法中。其主要规则——“程序的每一层都该自成一体”——在此后几十年里不断推动创新的产生，程序员们（怀着以可复用乐高积木式组件为最终圣杯的梦想）设计出各种新方法来孤立、“模块化”和“封装”软件中可移动的部分。然而，和这些将出现的创新一样，结构化编程并没有帮助那个时代的软件变得完美；反之，它为将来更具野心的软件埋下了坚实的基石，而那些软件也将以结构化编程想都想不到的方式遭到失败。

改进组织代码的方式，最终是与改进组织人员及其工作方式之间的步态竞赛[②]。当产业在整个 20 世纪 60 年代和 70 年代表现平平，新软件方法论的拥护者们的注意力也转向了企业中“人”的一方面。具体而言，在与让大规模项目搁浅的趋势做斗争的时候，他们开始聚焦于改进制定计划的过程。而且，随着时间推移，他们分裂为两个意见相左的阵营。就像开源项目的贡献者们反目成仇一般，他们也分道扬镳了。

其中一派看着计划说：“好计划太有价值，但是太难制定，你得更努力地做计划，花更长时间，更细节化，然后再做一次计划！直至无所计划为止——然后再做一点计划。”另一派说，本质上，“做计划是不可能的，而且结果也没什么用，你应该摒弃所谓的计划。船到桥头自然直！写代码、听反馈、问客户、不断修改。这是唯一值得依赖的计划。”坦白地说，我有点儿夸张。但这种分裂是真有其事，而且多数软件开发者和经理，因其天生倾向或痛苦经验，都会倒向其中一派。

项目管理大师瓦茨·汉弗里（Watts Humphrey）提出了“*Must.Plan.More!*（*必须制定更多计划*）”的主张：“除非开发者为个人工作制定计划并遵循之，否则工作将不可预料。而且，假使开发者的个人工作成本及进度不可预料，则团队工作的成本及进度也不可预料。当然，假使项目

[①]《圣经·旧约》中的一章，乃先知耶利米回顾他在《耶利米书》中预言圣城耶路撒冷被毁之事。

[②]旧时美国南部黑人比赛步态优美，优胜者奖以蛋糕。



瓦茨·汉弗里

团队的工作不可预料，则整个项目也不可预料。简言之，只要个体开发者不为个人工作制定计划并遵循之，项目就会无法控制、不可管理。”

另一方面，当代管理学之父彼得·德鲁克的研究结果却是：“关于知识工作者任务的多数讨论都建议他们制定自己的工作计划。听起来很好。可惜错在它很难实现。计划总是停留于纸上，想法总是好的。付诸实施者寥寥。”

德鲁克于 1966 年发表上述言论。与此同时，年轻的瓦茨·汉弗里即将执掌 IBM 的软件管理大权。



当时，IBM 正努力要推出 OS/360——IBM 新一代大型计算机 System/360 的操作系统——其惨败正是弗里德里克·布鲁克斯写出《人月神话》的肇因。在接受负责 OS/360 团队的工作之前，布鲁克斯已经一只脚跨出了 IBM 的大门；他正打算到北卡罗莱纳大学创办计算机科学系。依汉弗里所言，尽管布鲁克斯尽其所能带领 360 编程团队走过了概念和设计阶段，他还是在想法实现之前离开了，而项目也在他离开之后遭遇了“大麻烦”。

所谓**大麻烦**：缓慢。延误。再次赶不上重定的进度。更多延误。最终 IBM 交付了没配上完整操作系统的 System/360 硬件。光这就多花了整一年时间。

汉弗里于 1966 年 1 月接管 IBM 软件组织后，震惊于情况之严重，写下一份简报。“我首先要知道工作进行到什么位置……每到一个实验室，我就要求管理人员提供计划和日程安排。他们只拿得出非正式的笔记或备忘录。当我询问经理们如何看待管理软件项目的最佳方式，他们说应该在开发开始前做计划。当我问及为什么不这么做时，他们的回答

是没时间。这根本毫无道理！做这事的正确方法无疑就是最快和最廉价的方法。显然这些经理没在做管理，只是随机应变罢了。压力太大，工作太多，他们只能做那些发布代码所必需的事情。除了火烧眉毛的事，别的都往后推。我认为责任在我，与他人无涉。只要我允许他们在没有计划的情况下宣告和发布产品，他们就会继续这么干下去。”

汉弗里还发现，在软件团队还没有发布产品的计划或资源时，IBM 的市场部门就急于宣告。他跑去找老板。“我告诉他，既然所有的发布日程毫无价值，我有意全部取消。然后，我要叫所有软件经理为每个项目做计划。往后，在我拿到带有说明文档和开发经理签名的计划之前，我们将不会宣告、交付或投入资源到任何编程项目……那些实验室花了大概 60 天做第一份计划。这个从未做过交付进度安排的团队，在之后两年半的时间里再也没有延误过。”

汉弗里在 IBM 执行强制进度纪律的成功基于两条原则。计划是强制性的。计划必须符合现实情况——“从底至上”，依据那些负责按计划执行的程序员的经验和知识而来，而不是“从顶至下”，靠管理者拍脑袋或对市场的期望而来。

20 世纪 80 年代，汉弗里从 IBM 退休，考虑下一步应该做点什么。如他后来告诉《商业周刊》所言：“我女儿拽我去参加一个关于誓愿的训练班。在训练班上，一直在讨论 ‘发宏愿’，比如消灭世界饥饿。所以我也发个宏愿，就是离开 IBM 后，我要改变这世界的软件开发方式。安坐沙滩上，实在太无聊。”

汉弗里加入了卡耐基·梅隆大学软件工程学院（Software Engineering Institute, SEI）。SEI 是五角大楼于 1984 年成立的机构，旨在改进用纳税人的钱购买的不断增加的大量软件系统的质量。在 SEI，汉弗里和同事们创建了软件成熟度模型（Capability Maturity Model, CMM），作为一种衡量软件开发组织品质的准绳。

CMM 给出了可供编程团队攀登的五级台阶。要读完 CMM 原则的全部文档可能得耗尽你的余生，但汉弗里在 1997 年发表的非正式描述给出了简单的概述："位于第一级的组织基本上什么都没做。第二级组织做一些计划、跟踪、配置管理工作，也讨论质量保证之类的话题。第三级组织开始定义过程——如何工作、如何完成任务、可训练的事项等。在第四级，他们采用衡量准绳。他们有一套真正跟踪和管理自己所做工作的框架，一种可统计跟踪的系统。第五级组织拥有持续改进的过程。"

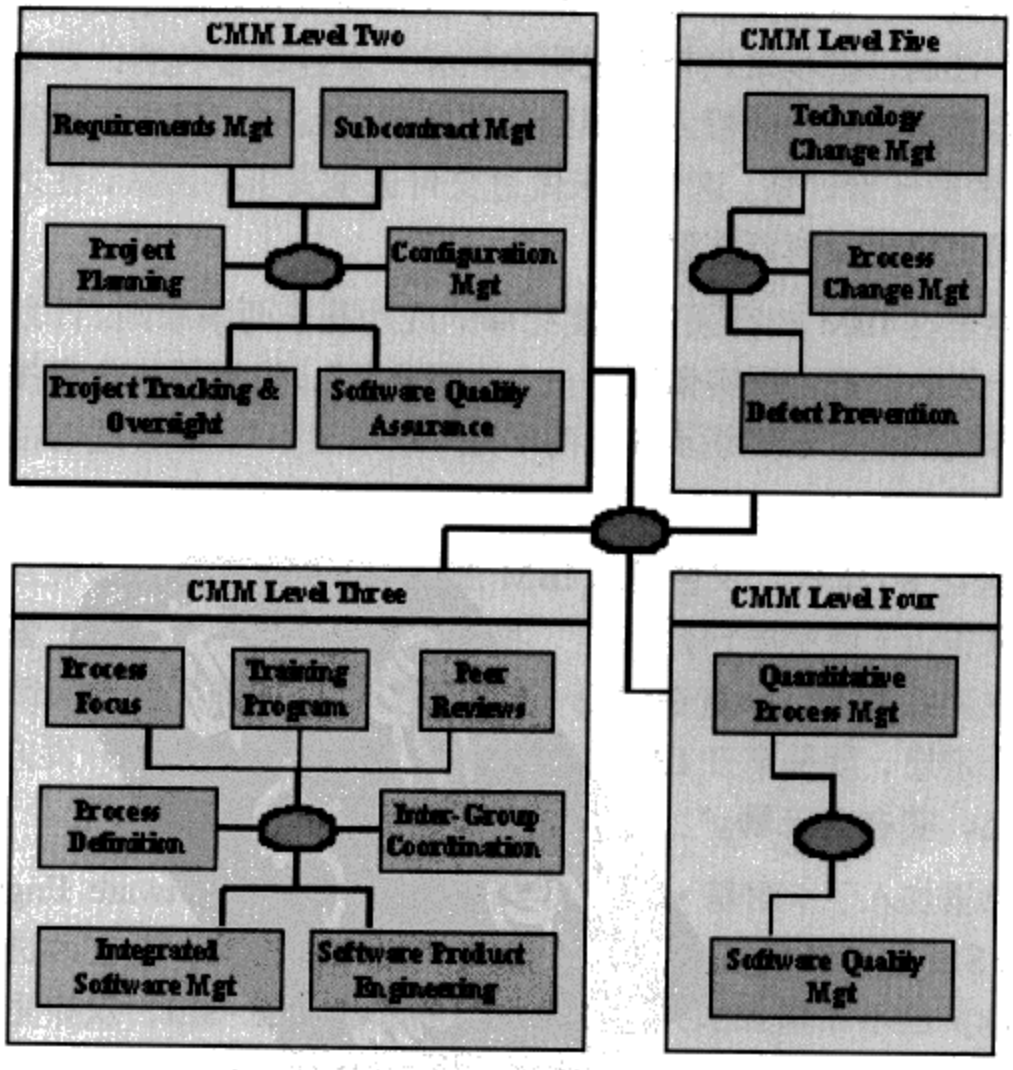


图 9-1　CMM 体系结构

美国国防部用 CMM 测量承包商的组织力量，它在这方面表现得最好。而在商业软件世界和急速增长的个人计算软件业中则问津者无几。直至今日，如果在多数小软件公司中提到 CMM，都会招来茫然的眼光。在关注过 CMM 的程序员和经理当中，最普遍的批评集中于其非人性和官僚化，带来沉重、缺乏创新的负担。在 1994 年一篇题为《CMM 不成熟之处（The Immaturity of CMM）》的文章中，软件质量专家詹姆斯·巴赫（James Bach）试图揭穿 CMM："CMM 重过程而轻人力……创新本身在 CMM 中没有得到体现，只在第五级上有所提倡……因为 CMM 怀疑个人贡献、无视需要非线性思维的情况、满足于将它们埋葬于约束的上层建筑之中，达到 CMM 2 级标准可能会扑灭公司赖以发展的星星之火。"

CMM 主要目的是帮助规模庞大的组织改进软件进度和质量。实际上，继 CMM 之后，汉弗里和 SEI 还提出一对相关计划——小组软件过程（Team Software Process，TSP）和个体软件过程（Personal Software Process，PSP）——某种程度上就是为了回应这些抱怨。TSP 和 PSP 批评"专制的管理风格"，鼓励个人开发者和小团队主动做计划和质量控制、分享信息、随需"动态平衡"工作量，以此夺回自己命运的控制权。汉弗里有一张幻灯片上列出了以下很有说服力的要点：

- 我们都为组织工作。
- 组织需要计划。
- 除非你的工作足够独立，否则必须按进度工作。
- 如果你不自己做进度安排，别人就会给你做安排。
- 这样别人就会控制你的工作。

CMM、TSP 和 PSP 的灵感都来自于制造质量专家 W·艾德沃茨·戴明（W. Edwards Deming）。戴明提出，质量不该是一种后悔药，而应体

现到生产过程的每一阶段上。对于软件而言，它意味着 CMM 反对“编码然后修正”（程序员先写出充满缺陷的产品、测试员找到缺陷、然后程序员再回头修正）的传统。“在所有的现代技术中，”汉弗里说，“软件是唯一一种在产品测试之前不考虑质量的领域。然而这不是工程师的错，因为他们就是这样被训练和管理的。”

就像戴明的理念在战后日本的汽车工业得到应用，远远早于在他的祖国——美国受到关注，CMM 也主要是在国外大受欢迎——特别是在印度快速增长的软件业中，软件公司发现，获得某个 CMM 等级会让海外客户认为他们值得信赖、足堪承担外包工作。但在美国的商业软件或桌面计算世界，CMM 及其相关方法论还没产生重要影响。

2000 年出版的一本名为《软件阴谋：为什么软件公司扔出错误重重的产品，它们如何能伤害你，而你又该怎么做（The Software Conspiracy: Why Software Companies Put Out Faulty Products, How They Can Hurt You, And What You Can Do About It）》的著作宣称这是一桩丑闻：既然这些方法被证明可以产生出更好的结果，为什么不是每个软件团队都采用、每个客户都要求呢？作者马克·米纳西（Mark Minasi）将指责范围扩大到不受约束的程序员、鼠目寸光的公司以及乐意忍受次品和缺陷的公众。

米纳西和其他 CMM 的拥护者认为，软件业知道如何按时、在成本预算之内生产出高质量的软件——但它又肥又笨又懒，不愿意接受改进的方法。他们也许说对了。另一方面，他们的言论假设软件制造者面临的最大挑战是降低产品中的缺陷数量。据汉弗里所言，在一位典型的有经验程序员的工作成果中，每 7 到 10 行代码就能找到一个缺陷；SEI 的研究显示，CMM 和相关的方法明显地减少了缺陷的数量。

CMM、TSP 和 PSP 借用了工厂管理的术语，将缺陷（bug）称作“缺点（defect）”，而缺点是被工人“注入（injected）”的——就像是说产品一开始处于某种柏拉图式的理想完美境界，后来被疯眼工人将充满缺陷

的针剂注入血管，遭到了破坏。这些方法强调一见到缺陷就杀灭，因为根据所谓的贝姆定律（Boehm's Law），在开发过程中越晚修正缺陷，代价就会越高。

“先修正缺陷”是个好原则，多数程序员都会同意该走这条路——原则上同意。但不管程序员抱着何种意图开始做，代码总是生而有缺陷，因为编写代码的过程，借用软件开发词汇表中最热门的说法，是“迭代”的。先写个能开始的东西；不断重复修正——“迭代”——改进。如果你非要坚持完美，就连第一行代码都永远写不完。实践上，在等着最终产品出来的人们失去耐心之前，“先修正缺陷”原则工作良好。然后，该原则就在忙于交付即便不完美也勉强可工作的代码的混乱之中被弃置道旁。

另外，缺陷总是存在。如果你瞅瞅任何一个活跃软件项目维护的缺陷列表，就会找到各种大问题和小麻烦。根据对用户使用的影响、寻找和修复缺陷的努力程度不同，缺陷也大有不同。而每份缺陷列表都是风马牛不相及的比较的具体表现。

软件缺陷编年史上记录了数量不多但足以令人警醒的惊人灾难。1962 年 6 月，水手一号探测飞船在发射后 5 分钟偏离轨道，为避免坠入居民区，飞行控制人员只好将其引爆。问题在哪儿？导航控制程序中少了一个连字符。1996 年 6 月，欧洲航天局投入 5 亿美元建造的阿丽亚娜 5 号无人运载火箭发射后 40 分钟爆炸，原因是在控制其导航系统的软件中有一个缺陷。（代码要将 64 位变量转换为 16 位变量，但数字太大，出现缓冲区溢出，系统停滞了。）从 1985 年到 1987 年，由于软件缺陷，一台名为 Therac-25 的放射治疗仪向 6 名病患放射了过量的 X 射线。在 1991 年的海湾战争中，在飞毛腿导弹袭来之时，爱国者导弹的一个电池未能成功点火；敌方导弹击中美军营地，导致 28 人死亡。调查发现，软件中存在一个累积计算错误，经过数百小时连续使用之后，爱国者导弹的计

数就会变得很大，从而无法点火。

上述事例及类似的故事数量太少，不足以引起媒体的愤怒和政治上的骚动，但它们却警示着我们对程序员工作质量的依赖。相对于每个致命缺陷的例子，都有上千个导致财产损失或经济损失的缺陷存在。人们指责软件问题导致了停电、航班延误和机场关闭、银行账户混淆、自动回拨，以及几乎所有你能想到的混乱和不便。

但引起灾难的瑕疵却并非“每十行代码一个”的缺陷；如果真是这样，那么任何软件系统都不能工作了。尽管计算缺陷数量明显是一种有价值的措施——总比不计算好——但也会倾向于把严重的系统故障和不值一提的小问题扔到同一个篮子里。而且还会鼓励程序员修葺既有产品，而不去打造新事物。这也不错——除非你就是想做点新东西出来。



在 20 世纪 80 年代和 90 年代，美国军方接纳 CMM 和软件工程学院字母形花片汤的其他部分，当作改进其所需软件的工具。但在千禧年之际，当五角大楼的软件规划师们考虑大规模未来新武器计划、同时面临后 9/11 军事承诺的压力时，他们开始担心，即便有这些方法论的帮助，程序员还是跟不上新系统所需软件的增长速度。

在 2004 年于盐湖城举办的五角大楼软件开发年会上，美国空军器材司令部工程和技术管理主任扬·奥格（Jon Ogg）指出，军方正面临他所谓的“软件发散困境”。在过去 50 年里，一个典型军用系统中的代码量增长了成百倍：例如，20 世纪 60 年代喷射战斗机可能会有 5 万行代码，而新的联合攻击战斗机则会有五百万行代码。但在同一时期，程序员的平均生产力只翻了一番。奥格说，这意味着我们发现“当今开发一种功能所需的人月数量”增长了 50 倍。

南加州大学软件工程教授、成本估算领域先驱和贝姆定律之父巴瑞·贝姆（Barry Boehm）在该次会议上以更简洁的语言指出："国防部需要软件的数量以指数级增加，而在有限时间内根本不可能完成。"

面临这样的困境，军用软件供应商们受到五角大楼这个最重要客户的鼓励，开始试验不太传统的措施。他们并不孤独。在 CMM 及其相关方法论在这些防务承包商中扎根的同一年，外界开始演化出另一种传统，它直接来自于那些知道自己的过程已经破毁、渴求找到更好路径的程序员的经验。

运动的一条主根埋藏于"模式社区（Patterns community）"中，组成该社区的软件开发者从建筑哲学家克里斯托弗·亚历山大处得到启发。20 世纪 80 年代末 90 年代初，面向对象编程的崛起及其令人生畏的复杂抽象把程序员们推向理解力的极限。软件模式运动——其领导者包括 wiki 发明人沃德·坎宁汉和一位名为肯特·贝克（Kent Beck）的程序员——为他们设想出一条新生命线，一种较少条条框框的软件方法论。他们没规定最佳实践的定规，而是以扼要叙述记录下他们的经验。"对于此类问题，"他们会说，"我们发现这种编程模式很有用。"模式运动将软件开发比作飞行器；如资深计算机专栏作家布莱恩·赫斯（Brian Hayes）所写，"程序员就是空乘人员，和木匠或石工一样，他们的知识自经验和师徒相承而来。"

坎宁汉和贝克等模式拥护者将上手实践总结带到迫切需要的领域。例如，对于降低某套面向对象代码的复杂性，坎宁汉和贝克建议程序员通过在桌上排布索引卡——每个软件对象一张卡片——来设计新程序。"卡片的好处是便宜、便于携带、容易获取、而且很常见，"他们写道，"我们惊奇于将卡片移来移去带来的价值。当学习者捡起一个对象，看起来更乐意和它打交道，而且也准备好从对象的角度来应付剩下的设计。正是这种物理性互动的价值让我们始终抵制卡片的计算机化。"

模式运动让个体程序员有了一条解决问题的新路子，并且提炼经验、供同事所用。但它并不太有助于找到在更大项目中协调工作的方法。

几十年来，典型的项目组织方式遵循“瀑布模型（waterfall model）”。瀑布方式——最初出现于1970年——将项目切分为依序进行的多个独立阶段，比如需求定义、设计、实现、集成、测试和发布。每个阶段需在下一阶段开始前完成。这套做法在纸上看似合乎逻辑，但实践起来却总是导致延误、混乱和灾难。每个阶段都耗时无算，但没一个工作正常的。程序员要么坐等需求，要么干脆在拿到需求之前开始做设计。“大设计优先”引发大延误，“大爆炸式集成”——单独编写代码的各个主要部分，在项目将近结束时拼到一起——导致系统崩溃。到产品终于完成那天，已经过了很久，该程序要解决的问题已经无关紧要了，新问题又在呼吁解决方案。

瀑布模型逐渐得到了它该得的坏名声。到了20世纪80年代中期，巴瑞·贝姆定义了一种叫做“螺旋模型（spiral model）”的替代方案，将开发切割为六个月到两年的“迭代周期”——更快生产出可工作代码的小瀑布，从部分完成的产品使用中得到反馈、指导下一步迭代。螺旋模型在大规模政府承包软件开发中成为标准，在那个领域，典型项目的“采办周期”可能会长达十年之久。

然而，在商业软件飞速发展的世界中，这还是太慢。20世纪90年代，软件业方法学的信徒们竖起了快速应用开发（Rapid Application Development）的大旗，RAD承诺通过快速原型设计和更紧迫的迭代周期，依靠新工具让计算机处理一些繁重的编程工作，加速完成软件的交付。RAD帮助软件公司更敏捷地工作——但在它开始掌舵后不久，就和Web的到来以及软件产业对速度的渴求再次凸显一起，迈出了互联网时间的疯狂步伐。

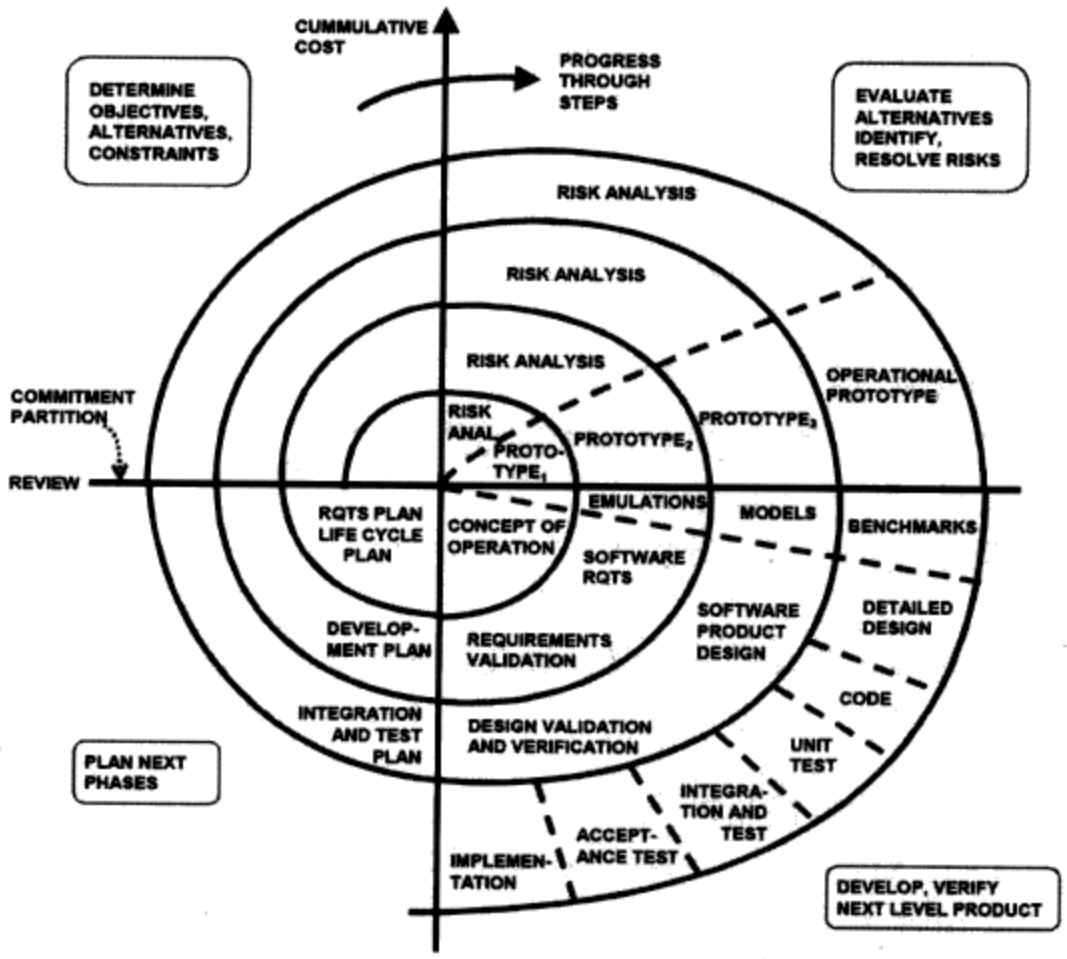


图 9-2　螺旋模型

20 世纪 90 年代快速出现的新方法论群体，钟情于适应项目的持续变化而不设法预测及控制其产出，被称作“轻量级方法论”。这让它们区别于 CMM 世界缓慢而笨拙的“重量级”方法论，却无助于将它们兜售给商业领袖们；它们听起来无所用处。2001 年，该领域中的 17 位领军人物[①]，包括坎宁汉和贝克，聚集于犹他州的滑雪胜地，想要找出他们互相关联但有不同的方法之间的共同立场。参加了那次聚会的软件测试方法论专家布莱恩 •马瑞克(Brian Marick)写道：“那次会议的部分目的……是找到术语‘轻量级过程’的替代说法。人们会觉得‘我是轻量级的’这种说法没有传递准确的信息。”

[①] 他们是 Kent Beck、Mike Beedle、Arie van Bennekum、Alistair Cockburn、Ward Cunningham、Martin Fowler、James Grenning、Jim Highsmith、Andrew Hunt、Ron Jeffries、Jon Kern、Brian Marick、Robert C. Martin、Steve Mellor、Ken Schwaber、Jeff Sutherland、Dave Thomas。

会议为运动取了个更有男子气概的名字——敏捷软件开发（Agile Software Development）——还发布了一份宣言，全文如下：

我们正通过实践和帮助他人来揭示开发软件的更好方法。

经由这项工作，我们估量：

个体和交互 胜于 过程和工具
可工作的软件 胜于 面面俱到的文档
客户协作 胜于 合同谈判
响应需求 胜于 遵循计划

即，尽管右栏条目有其价值，但我们更看重左栏条目。

在一个其理论文档倾向于过度使用缩略语和充斥八股文体的领域，《敏捷宣言（Agile Manifesto）》因其简明扼要而引人注目。但“敏捷开发”更像是共有价值观的保护伞，而不是具体过程的路线图。自从宣言发表后，各种相关但互异的敏捷方法论层出不穷。其中一种叫做争球式开发（Scrum），将项目分解为 30 天一轮的“竞跑”，强调每天开例会、维持项目在正轨上运转。不过，到现在为止，敏捷方法论最流行的变种还是极限编程（Extreme Programming，或称 XP——不要跟 Windows XP 操作系统搞混）。

肯特·贝克在 20 世纪 90 年代提出了极限编程背后的支持理念，当时他正为克莱斯勒公司（Chrysler）做咨询工作，试图挽救一套偏离正轨的薪资册自动化项目。“极限编程”这哗众取宠的名字听似跟批量购买高价运动用品有关，而且实际上贝克也是将“极限运动”对运动员专注和勇气的要求与极限编程对编码者的要求相提并论。不过极限编程的名称主要还是指它采用了一系列广为接受的方法，并将这些方法的使用推至极限。

“我们撷取最佳程序员已经采用的那些实践方法，并把所有的拨号盘都拨到 10，” XP 的创立者之一罗恩·杰弗里斯（Ron Jeffries）如是说。测试重要？那就让程序员在写代码**之前**写测试。开发组和客户交谈有好处？那就让客户在旁边随时回答开发者的问题。不做周期性代码走查，而是让开发者一直结对工作，这样每行代码从被写出来那一刻起都有不止一双眼睛看过。至为重要的是接受客户的需求，软件的目标就会保持变化，并且让项目“拥抱变化”。

XP 往软件开发世界中引入了一套全新并且有点陌生的词汇表。它要求将项目切分为“故事（stories）”：每个故事代表客户向开发者讲述的一种特性需求，解释了程序应该做什么。然后程序员为“故事”编码；如果结果正是客户需要的，万事大吉，准备做下一个故事。在 XP 激进的渐进机制下，不再有“大设计优先”；编码几乎就立刻开始。这对于许多急于开始编码但又受冗长设计过程所阻挠的程序员来说颇有吸引力。XP 认可的忘掉详细规约和代码文档的方式也很流行。

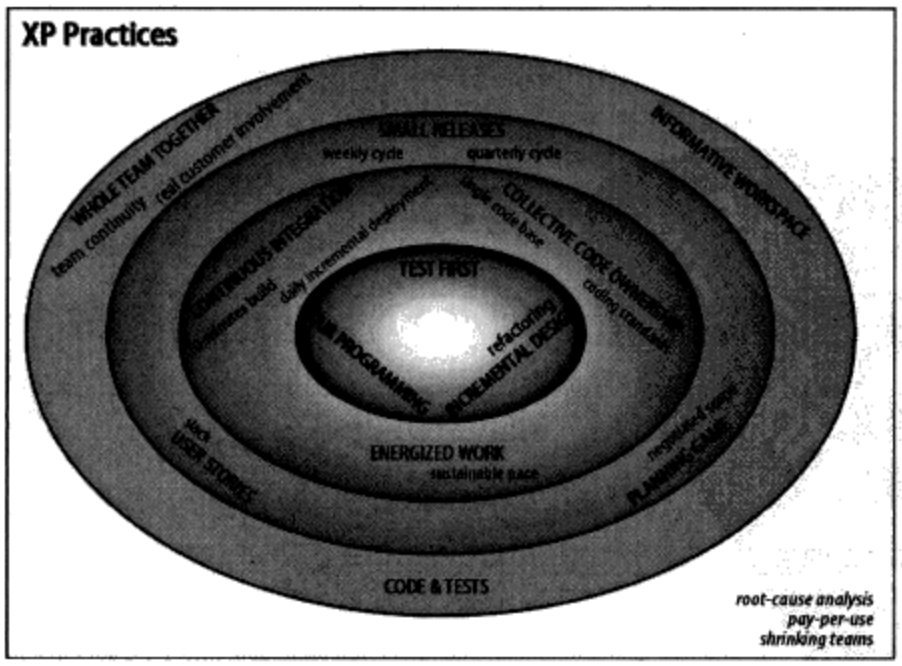


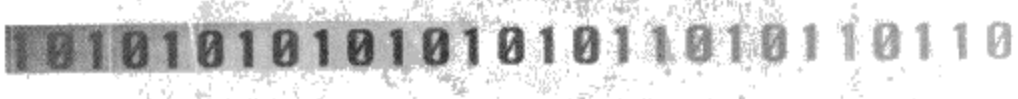
图 9-3　极限编程实践

但程序员也常发现有些别的 XP 原则难以遵循——特别是那句“你不需要它（You Aren't Gonna Need It，YAGNI）”的战斗口号。YAGNI 采纳了“刚好按时”的生产方法，采用这种方法的工厂在接到客户订单之前并不收集部件、制造产品。YAGNI 将这种方法应用到知识产品的世界。这条原则主张：“只在确实需要时才实现之，而不是在你预知需要时就实现之。”换言之：你的会议日程安排程序以后可能会支持位于不同时区的用户——但如果公司现在只有一个办公地，这个特性就放到以后再说吧。虽然管用，但对于编程部族中的磨斧头癖们来说实在痛苦。

一般公认敏捷开发和XP最适用于由有经验的编码者组成的小团队。尽管 XP 在整个软件业中都不乏信徒，甚至包括许多为五角大楼写代码的团队在内，但如果你想遵循所有 XP 规条，还是会感到过于苛求。这让有些程序员只遵循自己觉得舒服的规则。有批评意见认为，XP 常常成为编码者们偷懒的借口，他们借此逃避规约、文档、计划和所有“重量级”过程所要求的繁重负担的约束。用某位开源专家的话来说，XP 可能很快堕落为一种“快速交付废品”的方法。

有一次，在面试一位想加入 Salon 的应聘者时，我不无歉意地描述了当时我们技术工作的混乱状态。

“啊，”应聘者说，脸上露出了老练的微笑。“你们在用敏捷方法论啊！”



威利亚德（WILLARD）：*有人告诉我你完全疯掉了，而且你的方法也不太完善。*

克兹（KURTZ）：*我的方法不完善？*

威利亚德：*我真没看到……有什么方法可言，长官。*

——《现代启示录（Apocalypse Now）》

任何接触过这个领域的开发者都会承认，软件史过去 30 年中的方法论队伍丰富了这个领域，而且给了程序员以新工具和思考其工作的方法。但要想找到一个真正实施了某种特定方法论的开发人员或团队可不容易。有两位宾夕法尼亚州的软件工程教授在 2004 年做了一项研究，询问在多个行业中两百个软件团队的开发实践。研究报告写道，“我们震惊和失望地发现，最常用的实践就是什么实践也不用——有整整三分之一的受访单位采用了这种实践（如果可以叫做实践的话）。”

震惊和失望！许多软件开发者还在反抗，哭喊着“别把我关进去”。但其他许多程序员，尽管抵制那些标签和口号，还是静静地拥抱和推进了这些年里出现的各种方法论所建议的技术。“如果没有某种系统实践，这东西很难做，”他们会这样说。

祖尔·索伯斯基（Joel Spolsky）就是其中一员。他曾正式担任微软项目经理，负责 Excel 开发。Excel 是在 20 世纪 90 年代早期从米奇·卡普尔的 Lotus 1-2-3 手中夺得头魁并一直雄踞榜首的 Microsoft Office 电子表格产品。当索伯斯基离开微软、创办自己的小软件公司时，还开设了一个名为“祖尔谈软件（Joel on Software）”的网站，发表文笔尖刻但充满实用建议的趣文。以下是其中之一：

如果你还有哪怕一点点常识，就该问：“数据在哪儿？如果要我转用激烈编程（Intense Programming），我想要看到那些多花在狗窝和鸟笼上的钱这回要花在提升程序员自尊心上的证明。给我数据看看！”

当然，我们没数据……

你无法公正地比较两支软件团队的生产力，除非他们在完全一致的情形下做完全一致的事情，两边的人手也完全一致，他们以某种方式克隆出来，确保先前是白纸一张……

我们没有数据。你可以举出关于方法论 X 可行或不可行的各种奇闻轶事，但你却无法证明其可行不是因为团队里面有一位真正、真正优秀的程序员，而且你也无法证明其失败不是因为公司就快破产、人人自危、手足无措，即便是坐上安乐按摩椅（Aeron chairs）[①]也无所助益。

[①]网络泡沫时代的创业公司很多都购置这种按摩椅，极尽奢侈之能事。

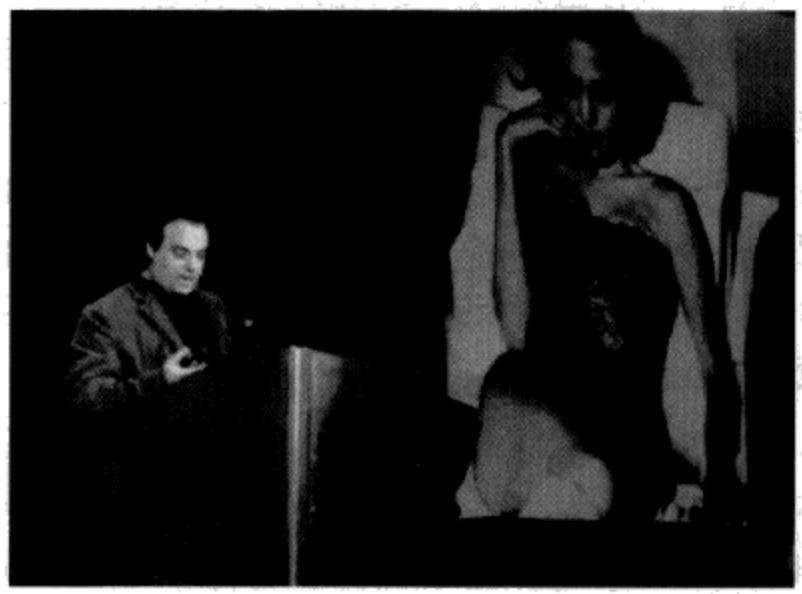


图 9-4　祖尔·索伯斯基的演讲和文章一样火爆热辣

索伯斯基非常怀疑他所谓的“巨无霸方法论（Big-M methodologies）”。在另一篇文章中，他写道：“小心那些方法论。它们是通往阴森但尚可通行的效能水平的康庄大道，但同时它们让那些痛恨被约束的才智人群极为烦恼。我很明白，好厨子不会乐意在麦当劳做汉堡，正是因为麦当劳那些规矩所致。”

索伯斯基写下上述文字几年之后，我采访他时，他仍旧心存蔑视。“方法论的真正目的，”他说，“是卖书，而不是真正解决问题。”索伯斯基还说：“方法论的关键问题在于，那类发明方法论的聪明人实施方法论时，就会有用。但如果是让那些只知道听令行事的笨蛋来实施，就不管用了。无论如何，大多数开发者不看关于软件开发的书，也不上关于软件开发的网站，甚至不看 Slashdot。所以，不管我们写出多少文字，他

们都领会不到。”

“即便到了今天，我环顾四周，虽然什么都上了网，还可以买到一百万本关于如何管理的图书，有一百万种关于如何做好软件项目的阅读材料，你仍然看得到同样的那些硅谷旧人物，他们本该知道得更多，因为这已经是他们开的第三个公司了，但他们还在犯同样的错误，做同样的错事，提同样愚蠢的假设。”

索伯斯基也有自己的一种方法论：他称之为祖尔测试（Joel Test），基于他自己的经验和“集体智慧”，给出一套快餐式原则，判断开发组织是否符合这条原则——“不会叫人头疼的CMM”。祖尔测试询问以下十二个问题：

你们使用源代码控制吗?

你们每步都做构建吗?

你们做每日构建吗?

你们有缺陷数据库吗?

你们会在写新代码之前修复缺陷吗?

你们有与当前工作吻合的进度安排吗?

你们有规约吗?

程序员工作环境安静吗?

你们采用了市面上最好工具吗?

你们有测试人员吗?

你们会要求应聘者在面试时写代码吗?

你们做走廊可用性测试[①]吗?

[①]到走廊上随便拉一个人试用程序。

“得 12 分为最佳，”索伯斯基写道，“11 分还可以接受，得 10 分或更低就说明你们问题大了。其实多数软件组织只能得 2、3 分，他们迫切需要帮助，因为微软等公司一直都得 12 分。”

诚然，微软是历史上最成功的软件公司，年复一年地赚取几十亿美元利润。它的行为值得留意。另一方面，与其商业竞争对手相比、或与那些编写公司内部软件的程序员相比，微软也不乏进度延误和忽视缺陷的情况。假使微软拔得祖尔测试的头筹，为什么还是有问题呢？“我想微软正处于在规程上要求万事齐备的阵痛之中，不再以一致水平制造人们愿意购买的产品，”索伯斯基说，“太过沉迷于过程……到了任何微软产品都要花 6 个月才能发布新版本的程度。他们真的开不了快船。在 SP2（Windows XP Service Pack 2）上花了差不多一年的时间。出于安全考虑，他们做了件好事，不过它的作用基本上也就是清理、维护和打补丁。”

“这就是军队叫做内务的东西。在军队里，内务就是保持装备在最佳工作状态的一切事情：擦鞋、刷牙、时刻准备着、子弹保持清洁、确保枪膛里没沙子。所有这些都叫做内务，步兵每天要花 2 个钟头做这些事。但它却不是你真正要干的事情。微软现在到了大概百分之八、九十的时间都在做内务的阶段。”

索伯斯基用步兵来做类比，这来自于他的第一手经验——他在以色列长大，曾经在那儿当过兵。不过还可以做更多比较。软件开发者常被描绘为激烈的公司战役中“死亡征途”上的步兵。他们接受任务，拼命在急剧变化的周遭环境和战争迷雾一般的可见度条件下完成任务。他们身上也常带有历史上老兵们那种宿命在天和坚定不移的组合特征。

我参加了那次五角大楼软件大会，聆听个体软件过程培训师戴维·库克（David Cook）半是鼓舞士气半是存在主义训诫地解释方法论规程的目的。

“你的软件有多少可能会改动呢？”库克激昂地问道，“噢，百分之

百会改。软件会用上 10 年左右。修改它的次数要比你想象中多得多。你会为了兼容各种软、硬件环境做改动，换 3 次语言，经历 4 次人事变动。你所能做的就是尽力保证质量，这样，在 10 年后你还能剩下点东西。”

“如同当你抓起软件，手中全是沙子。就在你站在那儿尽力而为的当口，沙子开始从指缝中漏出去。10 年后，只剩下几粒沙，那就是你要应付的事。你的任务是尽可能握紧拳头。”



尽管有那些可能性——复杂度、延误和不可预知的改动——还是有许多软件写出来、交付出去、而且最终被使用。偶尔软件会很好。在一些罕见情形下，软件的确有创新和价值。在一些案例中，还真**按计划**达成了目标。

在这些稀有案例中，成功往往是铁一般纪律的副产品——一种坚决做出又在每次遇到挑战时大声重申的选择，限制着项目的范围。在软件的成功故事中，你总能发现善于拒绝的人们。如同有意只在调色板上涂抹一种颜色的画家、宁肯写十四行诗也不写自由体诗歌的诗人，或者只固守小规模优势产品线的厂商一样，成功的程序员也在约束中成长，而非没有约束。有时候，约束是环境的产物——预算少、时间紧、目标有限。有时，约束是有经验的程序员或经理强加给自己的，他们懂得如何避开结局不可预料的——以软件界的说法来讲，“未绑定的”——项目。无论哪种情形，都更多地考虑“大即险”，而不是“小即美”。

有家位于芝加哥、名为 37 Signals 的小公司，正是这种拥抱限制的方式之代表者。37 Signals 最初是一家网页设计资讯公司，后来为了满足自身需求而将业务扩展到软件开发领域。他们编写了一些用于项目管理的内部工具。为了和客户沟通，就向客户开放了部分系统。公司创始人

和总裁杰森·弗瑞德（Jason Fried）解释说，在他们自己意识到之前，已经做出了一套基于网页的应用。又做了4个月，他们把软件转换为称作Basecamp的服务。Basecamp发布于2004年2月，很快在类似Flickr和Google的Gmail等新Web富应用天堂中名列前茅。

Basecamp只是这家公司花一年多时间投入少量程序员做出来的一系列值得注意的小而精的产品之一。Basecamp之后是Ta-da List，用于保存和共享待办事项（及类似事项）列表。几个月后推出了Backpack，它允许用户保存和共享便签及文件。每种产品都可靠并易于使用，而且都是精心设计的。每种产品通常也都只包括少量新特性。例如，Basecamp就有一些精巧的电子邮件功能：和其他服务和程序一样，也可以设置邮件到达提醒——还可以从另外的计算机或手机等移动设备向Backpack网页发送邮件，邮件文本就会在页面上显示出来。

我刚开始使用Backpack时，是用来保存本书的零散调研笔记。2004年秋天在一个技术大会上偶遇弗瑞德，我问他37 Signals怎么能在如此之短的时间内做出这么有用的软件。他大力鼓吹自己的方法——他公司开了个名为“制作Basecamp”的训练班，将所用原则做成了一套PowerPoint幻灯片——而且逼着我在酒店大堂里听了45分钟关于其方法论的概要介绍。

首先，37 Signals只有一位开发者，所以就避开了布鲁克斯法则的泥沼——就像米奇·卡普尔最初做Lotus 1-2-3那样，当时也只有乔纳森·萨赫斯（Jonathan Sachs）一位程序员。开发者之间的协调不成问题。37Signals唯一的开发者戴维·海因梅尔·汉森（David Heinemeyer Hansson）住在丹麦，就连这似乎也不成问题。弗瑞德说，在大多数公司里，地理上的分隔会被看做是严重问题，不过时差却让他们真的只有区区几个小时可以讨论，所以他们会高效利用这点时间，跟着开发者们就能平心静气地写代码，不受干扰。

照 37 Signals 的做法，约束是朋友。“约束是打造伟大产品的关键，”弗瑞德说，“约束产生创意。如果有人说，给你全世界的财富，让你做任何想做的东西，那这东西多半永远发布不了。给我一个月就好！”

37 Signals 生产优秀软件的另一关键要素是紧抓 Web 应用不放。所有东西都通过网页浏览器运行，所以程序可以在任何能运行浏览器的计算机和操作系统上工作。版本更新可以很容易地在运行服务的服务器上做到，用户无须下载和安装更新。汉森还热衷于 Ruby，一种面向对象动态编程语言。Ruby 近似于 Python，不过较少为人知，汉森发现它简化了自己的工作。最后，37 Signals 的方式还避开了编写规约的环节；相反，一开始就做用户将看到的详细网页。这些页面设计成了规约。弗瑞德说，他的团队很少会长时间争辩页面上的每个词、按钮和方块。

37 Signals 只做小程序，不做野心勃勃的新平台或应用程序框架。但在打造 Basecamp 的过程中，汉森还写了一些有用的创新代码，改善和简化了所有 Web 应用在保存和获取数据时都要执行的细节基础操作。Basecamp 发布后，他和 37 Signals 决定把这部分工作拿出来，作为一套开源平台发布，名字是 Ruby on Rails。这套将被命名为 Rails 的框架在某种程度上通过约束程序员的可选手段使得编写 Web 应用更为简单。“灵活性被过分高估——约束才是解放，”汉森说。Rails 也具备实现 AJAX 风格增强界面的能力，这种新界面风格让基于 Web 的程序足以与桌面应用抗衡。

37 Signals 从 Basecamp 中抽出 Rails 的同时，还从 Basecamp 的经验中归纳出一套设计哲学，体现为一系列小警句：“精简代码。”“拒绝在先。”“找对人。”“与其做半成品，不如做功能减半的优质品。”这些短句是为了通过幻灯片快速演示，不过合起来却是一整套软件开发方法——姑且称之为实用最小主义。它也许不能满足鼓舞了如此多程序员的改变世界之瘾。你也可以批评它是锋芒尽失的表现。它看似不适用于那些别无选择只能做大的软件。用程序员们的话来说，就是“配不上”。

不过依据 37 Signals 一直以来的跟踪记录，有个最大的推荐理由：它的行事方式看来的确有效。类似的方式在一家规模更大、也更为著名的软件公司中已经获得空前成功——甚至可以不太夸张地说，获得了改变世界式的成功。Google 遵循一种听起来很像杰森·弗瑞德推崇的那种软件开发哲学，成长为规模达数十亿美元的巨兽，并且开始挑战微软：每个新项目专设一个小团队，开发期限紧迫，做出目标集中的网页产品，然后再根据用户反馈和领域经验加以逐步改进。Google 也让开发者把五分之一的工作时间花在个人项目上。这“20%时间”的劳动成果可能会变成很酷的新产品——或者不会。不用担心，Google 安抚员工说：尽管开干，挠你自己的痒处。

Google 因打造了工程师天堂而获得赞誉，算法称王、编码者说了算。那些有幸受雇于 Googleplex[1]的人——包括安迪·赫兹菲尔德和 2005 年加入的 Python 发明人圭多·范·罗萨姆——暂时在这里逃离了软件时间的困境。Google 做出过一些半成品，但无人能质疑其成功的价值——从最初的搜索引擎到基于关键字的广告业务，以及流行的新免费电子邮件服务。

实用最小主义在 Google 用得很好。而且它现已成为一家声名显赫的公众公司，面对着跟上成长步伐和找到新收入来源的压力。对于许多人来说，看似 Google 正在一手制造硅谷的新泡沫。如果它在这种情形之下坚守其方法论，想出如何在不变慢、不变笨的前提下成长得更大，那么它将是软件业历史上独一无二的。

[1]Google 公司总部，来源于 googolplex（10 的古戈尔次方，一古戈尔等于 10 的 100 次方），Googleplex 也是科幻小说《银河系漫游指南》中的超级计算机。

1010101010101011010110110

我不能说自己对做更好软件方式的寻求获得了成功。尽管我不认为方法论传播者们全是假药贩子——他们都呈现了好不容易才获得的经验教训，软件制作者能从中获益良多——但他们也不免失之保守。他们给

出了解决困扰当今软件业的方案，但在下一波问题蜂拥而至时却无计可施，这也是为什么每过十年就会出现一大把方法论的原因。

然而在一种情况下方法论的确管用。那就是商业思考者尼古拉斯·凯尔（Nicholas Carr）于 2003 年 5 月发表在《哈佛商业评论（Harvard Business Review）》上那篇声名狼藉的以《IT 无关紧要（IT Doesn't Matter）》为题的文章中展示的场景。凯尔激怒了硅谷的幻想家和技术管理者，他提出，他们的产品——信息技术，或称 IT 的全部内容——已经变得无关紧要。当柏林墙倒塌、苏联解体时，黑格尔学派哲学家弗朗西斯·福山（Francis Fukuyama）[①]曾喊出了名噪一时的“历史终结（The End of History）”，现在凯尔也号称软件历史基本上已经**完结**——给它个痛快吧！我们知道软件是什么，知道如何在商业世界中部署软件。除了用重量级方法论使之完善外，别无他法。



图 9-5　尼古拉斯·凯尔：IT 无关紧要

[①]政治经济学家。日裔美国人，出生于芝加哥，分别在康乃尔大学和哈佛大学拿到了古典文学学士和政治学博士学位。著作品重点关注于民主化和国际政治经济体系等问题，主要有：《历史的终结及最后之人（The end of history and the last man 1992）》，《大分裂：人类本性与社会秩序的重建（The great disruption: human nature and the reconstitution of social order 1999）》等。

凯尔比较了过去几代的“破坏性技术”，例如铁路和电力，他认为一开始计算机和软件让具有远见卓识的早期采用者们拥有了获得相对优势的机会。他说，不过现在它们全都是商品了。它们很重要——忽视它们就会落后于竞争；但它们不能“让某个公司明显地与竞争对手区分开来”。在公司战略和商业竞争的大游戏中，它们无足轻重。它们变成了烦人的基础设施——只是些水管罢了。

凯尔文章颇具煽动性的标题引起了意料中的反驳，但如果你把 IT 看做一种功能和能力的固定集合，那么他的论点也不无道理。IT “无关紧要”，只要你一早就知道你期望用软件实现的业务——会计、工作流自动化、仓储管理等等——并不会改变，而且竞争对手也在使用软件完成同样的事情的话。

但在企业重金投入的所有资本货物中，软件是唯一一种多变的。那些占据了大公司 CTO 和 CIO 所有时间的用于“客户关系管理”和“企业资源规划”的庞大软件包，可能会相当笨重而昂贵。然而它们也还是——就像丹·布瑞克林和米奇·卡普尔在四分之一世纪之前引入到这个世界的小小电子表格——（用弗里德里克·布鲁克斯的话来说）“思想的产物”。所以，当人们决定将软件的某个部分用于新目的时，用到的部分就要进行改动。当你引入一套新的雇员调查系统，某人头上的灯泡就亮了——嘿，何不改造一下这套系统，用来安排公司晚会节目？只要几个星期就够！既然开始做了，开发组也许可以加一个讨巧的功能，管理我的音乐收藏——嗯，这样我们就能投票决定晚会上放什么音乐了！

我们是否已经发掘了软件提供的全部有用之处——当然，还有所有无用之处呢？相信这套说法，就不仅是傲慢自大，而且还是对计算机历史的贸然无视。例如，20 世纪 80 年代中期，商业分析师们断言整个个人计算机产业已经成熟并到达巅峰，可麦金塔和 Windows 风格的图形界

面系统即将出现。到了 90 年代早期，微软开始称雄于 PC 软件业，大家都以为整个业界趋于稳定——但互联网又将引领新方向。2000 年和 2001 年，互联网泡沫破灭，许多观察家都准备宣称整个网络工业可以盖棺定论了；然而就在街角，Google 成长起来，并且推动了在线商业的整个新一波浪潮。

一次又一次，那些打赌在技术工业的太阳底下不再有新鲜事物的人输了个精光。相信我们已经知道软件的所有可能用途，等于是假设我们拥有的程序能满足所有需求，人们也不会再去寻找更好的东西。

《软件阴谋》的作者马克·米纳西就是这类软件缺陷的愤怒批评者，与尼古拉斯·凯尔之类软件商业怀疑论分析者共有一副“历史终结”的眼罩。如果你相信我们已经知道需要软件做的所有事情，那么很自然也该相信，只要工作足够努力、计划足够详细，我们就能让软件变得完美——而且那也正是该全力投入的地方。不要再想新特性和新点子；所有人都集中力量于缩减缺陷列表，直至有史以来第一次可以说大多数软件已经备其全形。微软确乎尝试过；在一连串令人难堪的病毒爆发事件后，微软从 2003 年开始停止所有新产品开发，将大量编程人力投入到修正 Windows 安全漏洞的工作上。结果推出 XP Service Pack 2，改善了 Windows 安全机制，但这并没能挡住人们发明新病毒来折磨它的步伐。

这种孤注一掷、不顾其他的做法偶尔也有意义，当普遍使用的有缺陷软件亟待修正时尤为如此。不过在多数时候却无关紧要：它没明白为什么我们对自己依赖的软件还不满意。用户可能会为缺陷所烦扰，软件开发者可能会因没能做到最好而失望，经理们可能会因他们计划之不可靠而感到挫败。但是到最后，什么也不及软件不以我们期望的方式工作来得严重，而且只要它继续这样，就不值得去打磨。



一次在OSAF吃午餐时，约翰·安德森承认，有时他会觉得，如果能挥动魔杖、按计算机内存大小计将时钟拨回十年前，那么软件可能会更易于编写、质量也会更好。

“用今天的工具和过程，加上昨天的内存限制，我们真的能做得更好，”他说。

“你只是说说而已，还得设定软件目标的上限，”我反驳说，“应该做功能更少的软件。”

“嗯，对，”他答道，“功能应该更少——但缺陷也要更少。”



只要我们持续要求软件完成新任务、解决新问题，修正今天的缺陷就不能让我们免于明天的崩溃。在打算重复成功时，虽然软件方法论非常有帮助，你还是只能照葫芦画瓢。如果是在探寻未知的领域，最佳实践可能会帮助你稍微加快一点进度，但却无法指点前进方向。

软件开发常与建筑工业相提并论，但这种类比在某种意义上站不住脚。尽管我们学会如何建造解决特定问题的建筑——供一家人居住，或者为治疗病患提供合适的空间——我们还是得不停建造更多建筑。不管我多喜欢你的房子，我也不能据为己有——你已经住在里面了，如果我也想要，就得再建一幢。软件可不是这样；只要我们知道如何写一套程序来计算支票簿余额或显示一张网页，那这个程序的额外副本完全没有生产成本，你付出的价钱，如果你给钱的话，是为了奖励设计出软件的人，而不是为了制造这份副本。

所以，如果有人写过满足你需要功能的软件，那么买一套总比你自己从头写一套来得便宜。

然而，一次又一次，我们发现自己所需之物总与现有之物差异甚大，只能编写新代码。例如，在 Salon，20 世纪 90 年代末，我们意识到需要自动化发布文章。当时最流行的“内容管理”软件是一套叫做 Vignette 的大系统，价格高达 6 位数。该公司销售人员到我们办公室拜访，描述了软件的几百种有用特性。很不幸，其中大多数我们并不需要。另外还有一大把我们确乎需要、而 Vignette 却没有做到的功能。所以我们请了一些程序员，编写自己的系统。

所有的软件作家迟早都会提出一条以自己名字命名的法则，现在是我满怀谦卑地呈示罗森伯格法则的时候了：*软件好做，除非你想让它完成新任务*。接下来当然可以推论：*只有完成新任务的软件才是值得做的软件*。



“软件不过是水管”的论点引起了几乎是下意识的“没错，但是”反应。水管满足有限但定义良好的需求集；我们在很久以前就设定需求（有热水和饮用水、不漏、不堵的盥洗室），甚至当创新者试图改变他们——电动马桶座圈，谁想要？——改动经常都会失败。

迄今为止，在其简短但被拉长的历史中，软件都不是那样的。

现今软件的批评家们常常抱怨我们还处在这项技术的“修补引擎”阶段。在汽车业发展的早期，车辆很有可能在到达目的地之前抛锚，如果你想开到任何地方去，都得先做个业余机修工。同样，今天的计算机用户常常也得当业余程序员，学习机器的思维方式、修复在完成工作时遇到的大量问题。最终，汽车越来越可靠，控制器件越来越标准化，不

必为了驾驶而勉力成为汽修爱好者；可我们却一直在等待，期待按照同样的原则，软件领域中也能出现类似的情形。

此论有理。也许有一天我们能到达软件真的变成像水管或汽车一样的地步——能循常例生产、不费心思也能用、而且永不改动。

然而，若以过往为鉴，看来软件还是会做全新的不同工作。这个十年的汽车会在下个十年变成飞车——然后，潜水艇！而且我们会欢迎、甚至要求这样的变化。

第 10 章　工程师和艺术家

以下意见来自某次软件工程大会上的专家小组会：

“我们无疑是在用落后的技术制造软件。”

“特别令人惊心的是大软件中看来不可避免的错误，因为现代软硬件系统中的故障关乎生死。”

“我们像怀特兄弟造飞机一样做系统——做完之后，推下悬崖，坐视坠毁，重新来过。”

“大型软件生产已经成为一种管理上的恐慌事务。它常被看做是无利可图的沼泽地，既费钱又没完没了。”

“如果至少能预先对软件任务所需的人力和成本进行限制，这问题也不是那么吓人……没有理论可以用来计算对软件尺寸、性能和复杂度的限制。许多例子说明，甚至不能以符合逻辑的手段来说明软件产品要做的事情以及它如何做到的问题。”

“我想，用自然语言，比如英语，几乎不可能定义完全清晰的系统。”

“有些人认为，四、五个人在一年之内不能完成的软件系统将永远无法完成。”

这些严苛的观点听来耳熟。不过我省略了一个细节：这会不是最近开的。上述言论，以及另外上千句类似的话，都可以在 1968 年某次软件工程峰会的记录中找到——那次会议开类似活动风气之先，由北约组织举办，当时该组织各成员国都看到了他们开始称作“软件危机”的东西。

在经历了 IBM OS/360 灾难和其他问题重重的大规模软件项目之后，仍然身陷冷战、并将竞争延伸到外层空间的北约成员国决定将软件项目视作紧迫的国际问题。他们召集了几十位智囊到德国的嘉美善（Garmisch），一处幽静的巴伐利亚滑雪胜地，向他们征求对软件可靠性、质量控制、成本控制和进度安排的看法——我们今天将这些主题统称为“软件工程”。

大会召开后十年内，“*软件*”和“*工程*”两词已密不可分，就像“*政府*”和“*官僚*”、“*快*”和“*餐*”一样。在大公司和小创业公司里面，随便谁有本事搞出个脚本、把某个什么事务自动化，就能荣膺软件工程师的称号。这并非人人乐见之事。2003 年，一些自称软件工程师的程序员收到来自得克萨斯州专业工程师委员会（Board of Professional Engineers）的禁制信，信中不无嫉妒地阻止他们使用这个名号。撇除这些官僚卫道士的做法不谈，“软件工程”一词已趋通用，只偶尔有人质疑将修造建筑物与编写代码相提并论是否合理。

在 1968 年，那并不算是先见之明。据负责组织大会报告的英国软件专家布莱恩·兰德尔（Brian Randell）所言，嘉美善会议的组织者们以软件工程之名召开会议，他们“完全承认该术语……与其说是体现现实，

毋宁说是表达需求。为了蓄意挑拨引起争论，特意选择‘软件工程’一词，暗示了要求软件厂商基于理论基础和实践原则的需求，而这些正是工程学中已成立的各个分支的传统。”与会者打算找出将这混乱领域带到工程科学钟摆之下的路径；这是一种愿望，而不是业已实现之事。



图 10-1　智者云集的佳美善大会

然而在 1968 大会期间显然有事发生，让参与者群情激昂，并于第二年在罗马召开的会议上得以延续。第二次大会议题集中于软件工程的具体技术。“和第一次大会不同……在罗马会议上，有一种好像是说该主题（软件工程）已然存在的轻微倾向，”兰德尔在稍后的一份大会备忘录中这样写道。术语“*软件工程*”几乎是在一夜之间从“蓄意挑拨”变作了**既成事实**。

在大会的原始报告中，兰德尔写道，罗马会议“与前次会议少有相同之处。会议没有显现出嘉美善大会上那种共有问题的紧迫感。反之，至少是以编者之见，不同派别参与者之间缺乏沟通是主要特点。最后，这种沟通鸿沟的严重性，以及认识到它是真实世界情形的反应，导致鸿沟本身成为讨论的主要议题……”

有感于罗马会议上出现的分歧和关于是否需要成立国际软件工程研究所（International Software Engineering Institute）的争论，一位与会者——

IBM 程序员汤姆·辛普森（Tom Simpson）写下一篇小品文，题为《杰作工程（Masterpiece Engineering）》。辛普森假想了在公元 1500 年召开的一次旨在将艺术杰作“科学化”的会议。会议试图定下《蒙娜·丽莎（Mona Lisa）》的创作标准，并建立一间研究所，推进伟大画作的有效生产。“他们着手给杰作工匠们配备高效工具，帮助他们创制杰作，”辛普森写道，“他们发明出动力凿、颜料自动挤压器等等……生产仍未达到满意程度……研究所花了两个星期计量一组画家每天绘制多少笔，而且很快将这一标准应用到对其他人的评估上。如果某位画家一天画不了二十笔，那他就明显没达到生产力要求。很遗憾，这些知识进步对杰作的产出看来没什么作用，最后，委员会判定，难就难在管理上。他们很快就任命一位聪明的学生（名为莱昂纳多·达·芬奇[L. da Vinci]）担任项目经理，让他负责为组织的其他成员分派颜料、画板和画笔。”

辛普森对罗马会议的嘲弄不乏刻薄。“往后几百年内，”他写道，“有人会在此地挖出我们的录音带，发现我们同样荒谬绝伦。”这令会议组织者相当难受，他们迫使兰德尔和报告的其他作者把文章从会议记录中撤了下来（若干年后，兰德尔在网页上贴出这篇文章）。“杰作工程”在标记软件危机及其解决方案的讨论的中心断层线方面做到了极致——一方把做软件看作是一种科学过程，相信持续改进、甚至可能做到完美无缺；另一方将其看作是一种以创作为主的活动，虽然可以调优效能，但永远无法像钟表一样运转。

北约软件工程大会两份 130 页的报告实际上预示了将在后来四十年中占据软件领域的话题、观念和纷争。与会者记录下他们对于大规模软件的缓慢进度和不确定结果的沮丧心情，提出许多在后来的岁月里呼啸而至又消失无踪的解决方案：小团队。“在设计过程中及早获得用户反馈。”“现在就做点小而有用的东西。”“采用准则：必须易于解释。”

乔治·桑塔亚那（George Santayana）[①]的名言“忘记过去者必重蹈

[①]哲学家、文学家，批判实在主义和自然主义主要代表。生于西班牙，1872 年移居美国。

覆辙”应验了。想来所有程序员和他们的经理都该读过北约的报告。但据祖尔·索伯斯基说，多数程序员不怎么研读律己之规。这让他们反复身陷无明。



我们常视艺术活动与科学工作为绝不相干之事，但两者实有类同。科学与艺术以不同的比例分据了创作与洞见的多数工作；“杰作工程师”莱昂纳多·达·芬奇或许是这道理最著名的标杆。然而软件领域长期处于众多专制者的诅咒之下，他们坚称全部问题的解决之道在于摆脱人类的不精确性，要将其隔离于纯科学的净土之中。最极端时，这种观点的支持者强令以数学方法验证我们的程序；唯有“被证明正确”的程序方可视作完成。我没发现有多少实践程序员看重这种做法。（而且，如下一章所示，它也许在理论上也完全不可能做到。）

工程师当然要在艺术与科学的深渊上搭起桥梁。工程常被定义为将科学原则应用于满足人类需要。但它也让科学原则背上创意的负担，将它们从质朴的抽象里拉到挫折与愿望的妥协宇宙中。“工程（engineering）”一词由法语转逶而来，与“独创（ingenious）”同出于一个拉丁词源，指巧妙制造的能力。如今我们在多种场合使用“工程”一词，还将它绑到别的词上，表示一系列学科分支：机械工程、土木工程、电气工程、结构工程、化学工程等等。软件工程看似不过是名单上的后来者罢了。那这个概念为什么问题重重呢？

如我们所见，软件有时感觉难以对付，是因为它不可见。不可见之物难以设想，而高度复杂的不可见之物尤难用作沟通话题。但是不可见并非唯一问题：我们也看不见电力、磁力或重力，但却能为多数实用目的可靠地预测其行为。可是，想要从软件中取出一块行为保持一致的部分以便诊断问题，这却是我们常常无法做到的。

在好多网络幽默文集中，你都可以看到下面这个笑话。（文中关于阿尔卑斯山的部分甚至有可能是在向北约嘉美善会议致敬。）

软件工程师、硬件工程师和部门经理驾车去瑞士开会。行驶到一处陡峭山路时，刹车突然失灵。汽车不受控制，一路侧滑下去，飞越过紧急缓冲障碍，奇迹般地蹭着山石停了下来。乘客们有惊无险，不过面临一个问题：他们抛锚在半山上，汽车制动无效。该怎么办呢？

“我知道怎么办，”部门经理说，“先开个会，提出愿景（Vision），形成任务书（Mission Statement），定义一些目标（Goals），持续改进（Continuous Improvement）并找到严重问题（Critical Problems）的解决方案，这样就能上路了。”

“不行，不行，”硬件工程师说，“太花时间了，而且，这种方法从来就行不通。我带了把瑞士军刀，转眼间就可以拆下汽车制动系统、分离故障并修好它，这样就能上路了。”

“嗯，”软件工程师说，“动手开干之前，我想应该把车推回到山上，看看事故是否会重现。”

如果向程序员报告缺陷，他的第一反应是问你，“重现问题了吗？”——意思是说，你能确实让问题再现一次吗？如果答案是肯定的，事情就成了一半。如果答案是否定的，程序员就会耸耸肩膀，将责任推给硬件故障或者宇宙射线。

“‘软件工程’是一种矛盾修辞法，”在20世纪七八十年代服务于传奇般的施乐帕罗·阿尔托研究中心的软件老手L·彼得·德奇（L. Peter Deutsch）曾这样说过，“没有物理学就很难有真正的工程，而软件中则没什么物理学可言。”

其他门类科学的学生时或被指有“物理羡妒”，如计算机先锋艾伦·凯所言，这是因为物理学者“用严格的数学方法来对付宇宙的抽象基础”。对“软件物理”的搜寻成了软件研究者们数十年以来的追求。假使我们能揭示软件运作的可靠原则，就能超越鲁德·古德堡（Rube Goldberg）[①]式程序的高度复杂性，趟出一条走出软件时间泥潭的路子。终有一日，我们会发现自己处在一个软件无须编程的世界中。

对于许多搜寻者来说，共同的圣杯就是“自动化软件”的概念——即非程序员也能创作的软件，命令以计算机能理解的简单英语写成。这美梦诞生于计算机时代的黎明时分、海军少将格瑞丝·胡珀（Grace Hopper）[②]及其同事发明编译器之时。“胡珀相信编程不必是一种困难的任务，”她的海军官方简历这样写道，“她相信，程序可用英语写就，再翻译为二进制代码。”胡珀设计的 Flow-Matic 语言是 Cobol 语言的根源之一，Cobol 后来成为直至千禧年之际[③]仍广为采用的商业编程语言。



图 10-2 海军少将格瑞丝·胡珀

[①]美国著名漫画家，曾以政治漫画获普利策奖，但他最出名的还是所谓 Rube Goldberg's Inventions 漫画，表现用极复杂的机械组合完成极简单的工作。鲁德·古德堡式程序的意思是程序内部非常复杂，但要完成的任务却较为简单。

[②]计算机史上最杰出的女性，Cobol 语言雏形的发明者、第一个编译器 A-0 发明者、术语“bug（缺陷）”发明者，长期为美国军方工作，授衔海军少将。

[③]胡珀当年出于节约内存用 6 位数表示日期的设计，随着千禧年的迫近，终于引发危害巨大的“千年虫”问题。

相对于计算机语言 0 与 1 的二进制代码，胡珀的编译器进步巨大，它为后来的所有软件进步奠定了基础。但书写 Cobol 或任何其他编程语言代码和书写英语仍然是两码事，这并非对胡珀工作的毁谤之词。同样，Fortran 的发明者们本希望这种突破性的语言可以“本质上消灭编码”，却在有生之年看到自己的创造因其粗陋而受挫。

但从本质上消灭编码的梦想以好几种形式延续下来，培育了许多新缩略语，如同藻花按字母顺序绽放一般。有 UML，即统一建模语言（Unified Modeling Language），为独立于特定编程语言的绘图及设计软件提供了一套抽象词汇表——盒子、气泡和箭头。有 MDD（Model Driven Development，模型驱动开发）和 MDA（Model Driven Architecture，模型驱动架构），表示围绕 UML 进行开发的做法。有“生成编程（generative programming）”，最终软件产品不是人写出来的，而是由生成器（实则是另一套程序）生产出来。还有较新的时髦词“软件工厂”，在亨利·福特（Henry Ford）式的将软件构建过程自动化旗帜之下融合了上述一些概念。

如今在实现消灭编码的梦想方面，最引人瞩目的项目是由在施乐和微软发明了现代文字处理技术的查尔斯·西蒙尼所领导，前面已经写过，此人是匈牙利语标记法之父。西蒙尼在微软效力多年，带领一项在他称之为“意识编程（intentional programming）”领域的研究项目。2002 年，他离开了让自己成为亿万富翁的公司，投资一家名为意识软件公司（Intentional Software）的新企业，意在将研究转换为真正的产品。

西蒙尼是位潇洒而魅力十足的男士，他在十几岁时就逃离了匈牙利，但仍带有匈牙利口音——胸怀宏图。他和地球上的其他人一样对软件难题长久而悲哀的历史有着清楚的认识。“据我们所知，软件是数字化大丰收的瓶颈，”他说。摩尔定律驱动计算机芯片和硬件的速度和效率以指数曲线增长，但软件却步履蹒跚、赶不上趟。

在西蒙尼看来，问题的根源之一是我们对程序员指望太多。我们要他们成为从非程序员处提取出知识的专家，想出要做什么；我们要求他们成为把这些愿望讲述给计算机听的专家。“软件设计有两个意思，”他说，“其一是我们要打造的产品。其二是让产品得以实现的软件工程。我相信有两种不同的角色——主题专家和软件工程师。”



年轻时的查尔斯·西蒙尼

意识软件公司旨在“赋权予”那些“主题专家”——那些理解软件应用领域的人们，不管是空中交通还是卫生保健或者冒险游戏。西蒙尼想给这些主题专家们提供一套工具，他们能用这套工具以一种电脑能理解的结构化方式解释自己的意识和需求。意识软件公司将让非程序员专家定义一系列问题——例如，对于一套医院管理系统，他们可能会以一种机器可读的格式编列出所有“施动者”及其“角色”、要实施的任务和所有其他细节。然后把这些定义，即模型，喂入一套生成器系统，吐出最终软件。程序员还得花些力气为主题专家们创建工具、编写生成器。不过一旦工作完成，非程序员就能修改模型、改动软件，再也不用“低三下四求助于程序员”了。看呀，妈——不用编代码啦！

技术记者克莱尔·崔斯兰（Claire Tristram）在一篇描写意识软件公司所做工作的文章中打了个比方：“这就像是建筑师能画出一张有魔力自动造出建筑物的蓝图——甚至蓝图修改后建筑也会自动重建。”

西蒙尼将软件开发过程的复杂性自动化之梦是这个领域中高瞻远瞩的思想者们共有的梦想；不同之处在于，他拥有将其实现的决心和财力。“有人问过我，应该用什么语言来编写百万行代码的程序，”西蒙尼在2005年一次采访中说，“我问他，怎么会想要写百万行代码的程序呢——是因为有长达一百万行的问题吗？百万行的东西是大英百科全书——全书20卷，每卷上千页。很难想象一种业务实践或是一个管理问题里面会有这么多细节。”西蒙尼认为，你只需用一万行代码做“问题描述”，再用一万行代码做生成器——这两万行可管理性更强的代码包含了与那百

万行代码的怪物“同等的信息内容“。

年轻时，西蒙尼领导了 Bravo 的开发。Bravo 是第一个以现在被程序员们称作 WYSIWYG（What You See Is What You Get，读作“wizzy wig”，意为“所见即所得”）的方式运行的字处理程序。今天，我们理所当然地认为文档、包括其字体字号和图片，在打印出来时都该和在计算机屏幕上看见的一模一样。不过在 20 世纪 70 年代中期，那可是场革命。

从某种角度上看，西蒙尼的意识软件是在尝试将 WYSIWYG 原则应用到编程行为本身。但西蒙尼对由他的发明创造出来的全新软件世界的狂热描写却留下了一个核心问题待人解答。意识软件公司会让主题专家们灵活地直接对机器阐述他们的需求——还是要求非程序员专家自己戴上紧箍咒、创建一种特别细节化、“机器可读”的模型？它能帮助计算机更好地理解人，还是迫使人像计算机一般沟通？西蒙尼非常乐观。但截至笔者写作本章为止意识软件公司什么产品也还没发布，所以结果尚难以逆料。



我们已经看到，软件是一种分层结构，每一层都为相邻的上下层翻译信息和过程。底层是机器，只有纯粹的二进制 0 和 1。顶端是人类，人类打造和使用这些层。西蒙尼的意识软件本质上是在人机之间多加了一层。

层级是软件的精髓，驱动了软件领域的进步，但它们身患痼疾。它们有漏洞。例如：Microsoft Windows 许多版本的用户见惯了死机蓝屏现象。你正用着网页浏览器或 Microsoft Word，突然间，屏幕毫无缘由地变蓝了，上面写着些白字：

A fatal exception 0E has occurred at 0167:BFF9DFFF. The current application will be terminated.
（致命错误 0E 发生于 0167:BFF9DFFF。当前应用将被中止。）



图 10-3　致命错误蓝屏

看到单色屏幕和难看的字体，老资格用户会感觉在计算机时间中闪回到过去。有些人甚至会明白，屏幕信息所谓“致命错误”意思是程序遇到无法恢复的缺陷、导致崩溃，那串神秘的十六进制数字描述了计算机内存中程序崩溃的确切位置。这些信息对多数用户毫无价值。他们正用着的舒适、熟悉的应用界面消失了；更深一层的抽象——在本例中是 Windows “外壳”或较低层的控制程序——陡然爆发，就像是倾斜的基岩穿过较新的地层、在阳光下喷薄而出。（虽然死机蓝屏令人困惑，它其实已经代表了从早期 Windows 版本以来的重大进步，因为用户有时还能关闭出错的程序、继续工作。在蓝屏出现之前，单个 Windows 程序崩溃总会拖死整台机器和所有程序。）

在一篇题为《漏洞抽象法则（The Law of Leaky Abstractions）》的文章中，祖尔·索伯斯基写道：“在某种程度上，所有非平常的抽象都有漏

洞。抽象总会出错。有时少，有时多。漏洞与出错在所难免。”

对于用户来说，这意味着计算机有时会变得古怪而令人迷惑，有时你会想要像米奇·卡普尔在《软件设计宣言》中所说的那样，将它扔出窗外。对于程序员来说，这意味着那些捆扎底层计算复杂性、并将其打包到较新而易于操作的抽象层中的新工具和新概念很不错，但只在它们崩塌之前有效——然后所有隐藏的复杂性都漏回到他们的工作之中。理论上，就手的新顶层让程序员们可以不用理会下面乱七八糟的东西；实践上，程序员仍旧需要了解这些乱七八糟的东西，因为最终得立足于此。索伯斯基写道：

> 抽象并未真的像人们打算的那样简化我们的生活……漏洞抽象法则意味着，无论何时有人拿出一套本该能提升我们的效率的所见即所得新代码生成工具，你总会听到许多人说“先学会怎么手工做，再用所见即所得工具节省制作时间”。所以，抽象节省了工作的时间，却没有节省学习的时间……总而言之，尽管我们拥有了越来越高级的编程工具以及越来越好的抽象，但要成为编程高手却越来越困难了，这真够扯的。

所以，尽管“这些年来创造的抽象模型让我们能够应付十年或十五年前无须应付的软件开发复杂性新秩序”，尽管这些工具“让我们能极快地完成许多工作”，索伯斯基写道，“突然有一天，我们需要找出抽象出现漏洞的地方，而这需要两周之多。”

漏洞抽象法则解释了为什么当我和许多程序员谈及意识编程或类似欲图超越软件复杂性的概念时，他们都怀疑地转着眼珠。他们不是不肯踏上更高一级的抽象之梯；只不过他们害怕不管爬多高，总不得不违心地上上下下——而且，爬得越高，路程也就越长。



如果和程序员谈论抽象层级足够长时间，你多数会听到“乌龟一直向下掉落”的说法。这说法源自于一个流行的——显然乃是杜撰，至少来源不确——故事，斯蒂芬·霍金（Stephen Hawking）[①]用这个故事做了《时间简史（A Brief History of Time）》一书的开篇：

有位知名科学家（有人说是伯特兰·罗素[Bertrand Russell][②]）做了一次天文学讲座。他讲述了地球如何环绕太阳运行，太阳又如何环绕所谓我们银河的广阔星空运行。讲座即将结束时，后排一位老妇人站起来说：“一派胡言。这世界是巨龟背上的一片平地。”科学家傲然一笑，然后作答：“那么，巨龟又站在什么上面呢？”“年轻人，你真聪明，很聪明，”老妇人说，“不过那乌龟只是一直向下掉落！”

对于宇宙学家来说，这故事是讨论终极本源的好谈资。但对于程序员来说，意义就不同了：它提供了一种谈论内部一致系统的方式。比如，软件产品是“一直向下的对象”，或者“XML 一直向下[③]”、拥有统一的特性、无须经过复杂和容易出问题的内部翻译。

但对于那些读着苏斯博士（Dr. Seuss）[④]图书长大的人，比如我，“乌龟一直向下掉落”有另一种意义：它让人想起《乌龟雅图（Yertle the Turtle）》的经典故事。在故事中，龟王爬到乌龟们堆罗汉搭起的塔顶，查看王国的广阔土地。最后，塔底的一只乌龟打了个嗝儿——龟王就掉下来了，顺着其他乌龟一路滚过。“雅图”的寓意——嗯，有太多寓意了。对于程序员来说，教训可能是乌龟塔，或者抽象层级，即便有一小部分出问题，也会大受影响。用软件世界抱怨者常用的话来说，*太易碎*。承

[①]著名数学家、理论物理学家，英国剑桥大学应用数学和理论物理系终身教授。

[②]伯特兰·罗素（1872-1970），20世纪最有影响力的哲学家、数学家和逻辑学家之一，同时也是活跃的政治活动家，并致力于哲学的大众化、普及化，1950年获诺贝尔文学奖。

[③]指向下兼容。

[④]苏斯博士是Theodor Seuss Geisel（1904-1991）的笔名，他是美国家喻户晓的儿童文学作家和插图画家，作品包括《戴帽子的猫》等。



乌龟雕图

受压力时，不会弯曲——只会破碎。

“如果工匠以程序员做程序的方式建造房屋，来一只啄木鸟就足以毁掉文明社会，”计算机-程序员心理学先驱杰拉德·温伯格这样写道。水手 1 号的建造者们领教到，少个连字符，火箭就失常。

在 UML 的支持者查尔斯·西蒙尼和许多其他软件乐观主义者看来，只要梦想着往今天的龟塔上再加一层就足够了。他们指出，该领域过去就是这么发展的，何不继续下去？不过在今天的软件领域中也有一些批评者，他们相信问题就出在堆塔上。易碎性来自于我们对一堆不可靠概念的依赖。整个编程领域过去曾错转过几个弯。我们也许该重新来过。

01010101010101011010110110

艾伦·凯在满聚的听众前坐到笔记本电脑后面。他演示了一张幻灯片。幻灯片上，一条代表时间的上升曲线连接着两张相片。在曲线末端，右上角的相片展示了一座垮塌的桥梁，旁注“你在**这里**学到……”；在曲线开始处，左下角的相片是正在修建的桥梁，旁注“……本该在**这里**就知道的事情”。



艾伦·凯

凯，这位个人计算机开发领域的中心人物，他不是在讲与马后炮同工的教训。幻灯片的标题是“什么是延后绑定？”，而这个问题的答案如下所示：“延后绑定能够更早地进行变动。”

*延后绑定*是计算机科学中的一个术语，表示编程语言提供给程序员以更多灵活性的能力。延后绑定程序可在运行时修改；你甚至可以做出在运行时**自行改动**的程序。虽然时下流行的编程语言——Java 在某种程度上支持、Python（Chandler 采用的语言）提供更多支持——被看做是拥有延后绑定特性，但奠定现代软件大厦基础的核心语言 C 和 C++却不支持延后绑定。据凯说，C 和 C++不支持任意一种真正的延后绑定系

统——非程序员也易于操作的系统，很简单就能修复漏洞，功能也可改变。凯说，要找到具备延迟绑定动态威力精髓的语言，你得回溯几十年——回到人工智能研究者约翰·麦卡锡（John McCarthy）于 20 世纪 50 年代发明的“列表处理器（list processor）”Lisp 语言，回到凯自己于 20 世纪 70 年代在施乐 PARC 设计的 Smalltalk 语言。

然而，今天的程序员们仍备受“编译周期”的奴役。每当做出修改，就得等着系统消化它。首先通过编译器运行新代码；然后，重启系统，看看修改是否达到目的。如今较快的计算机缩短了编译周期，但程序员们还是无时无处不在咒骂它。Chandler 团队也不例外。尽管 Python 提供了一些延迟绑定的特性，但他们用来打造图形用户界面的工具仍旧依赖 C++；即使 Chandler 自身无须重新编译，程序员们仍旧面临一种修改并重启的周期——在看到代码修改效果之前，得等上三十甚至六十秒钟系统才能重新启动。积少成多，花去了不少等待时间。

凯普通的身高与其广博的学识相映成趣，近年来，他像一位长着胡髭的小个子耶利米（Jeremiah）[①]，在计算机的土地上游荡，指点技术发展水平，浩叹过去四分之一世纪以来该领域的错误走向，呼吁众人放弃他们的堕落之路、重新开始。“我们还不懂得如何做软件，”他说，“计算机人自满于他们做到的，而不是他们该做的……人们不明白还有替代方案。我们需要数百倍优秀于此的软件。”

20 世纪 70 年代在施乐 PARC 时，凯造出了“面向对象编程”一词，发明了重叠窗口界面的概念，还尝试实现他在 1968 年想出来的 Dynabook[②]——一种他梦想能为孩童想象力提供终极游乐场的膝上型电脑。在后来的几十年里，他看到硬件产业渐渐交付出和他想法相似的计算工具——但在他看来，软件领域却停滞不前。

在谈及软件时，凯喜欢采用历史类推的方式。“如果你看看当今的软件，”他在 2004 年对一位记者说，“那当然是某种工程——不过是那种没

[①]据《圣经》，耶利米是上帝指定的先知，先后预言了犹太灭国、埃及被巴比伦击败、巴比伦遭报应等噩讯，后被犹太同胞用石头打死。

[②]Dynabook 是现代笔记本电脑的最早雏形。

有建筑意识的人做的工程。今天的多数软件很像是埃及金字塔，堆积了上百万块砖头，却毫无结构整体性，只凭蛮力和千万个奴隶堆成。"

凯说，可以照这样构建大东西，但这"做不得准"。说到底，质量跟着数量变；在某一规模上可行的设计，到另一规模上就会失败。1997 年，在 Smalltalk 诞生 25 周年庆典上，凯略述了这一观点的逻辑。"随便用什么都可以搭间狗屋，"他宣称，如果把尺寸扩大一百倍，就会有麻烦了：质量增长以指数级别快于强度提升。

将某物放大一百倍，其脆弱程度也随之放大一百倍。如果胆敢如此这般建造一座大教堂，它只会坍塌成一堆残垣碎瓦。最常听到的回应说，那就是我们本来要做的事：堆上更多废料，刷上石灰加以掩盖，然后说，对，我们原本就在建造金字塔，而非哥特式大教堂。窃以为这也体现在今天的许多现代操作系统结构中……或者你能举出一种新概念，多年以前那些属意于复杂结构的人们也是如此开始的。他们称之为架构——字面意思是设计和建造出好拱门。简单材料之间不明显、非线性的互相作用，最后得到的是特殊的协同效果以及材料的大量增加。

凯说，每次他告诉别人查特尔大教堂（Chartres Cathedral）[1]的尖塔比帕台农神庙（Parthenon）[2]用的建筑材料要少时，人家总是大吃一惊："由空间和玻璃组成，以一种精美巧妙的结构组合到一起，整体统一性优于其中任何一部分。"凯认为，今天的软件学科其实尚处于其中世纪某处——"我们不必建造金字塔，我们可以建造哥特式大教堂，用较少材料造更大的建筑物"——但商业软件世界仍停滞于金字塔模式。

然而，他没有提议止步于大教堂。他相信应该有真正的计算科学；只是我们还没发现罢了。"总有一天，"他说，"我们会创造软件工程，所

[1]位于巴黎西北的查特尔城，建造于公元 12-13 世纪间，以两座哥特式高塔（分别为 114 米和 107 米）著称。

[2]位于希腊雅典卫城遗址中心，建于公元前 5 世纪。

有人的境况都会好起来。”他认为，这些发现和创造可以从细胞生物学中得到启发。他说，最终我们需要停止编写软件，学习如何种植它。凯提出，我们常把计算机系统看作具有不可管理的复杂性，但生物学家——他们常要应付拥有大量变动部分的系统——另有看法。对于这样的观察者，“不可能视计算机之属为特别复杂、庞大或快速之物。缓慢、细小、愚蠢——这就是计算机。”

凯关于面向对象编程——简写为 OOP，他念作“oops”一词省去那个“s”——的最初图景要比只是把代码组织为可复用例程壮阔许多，但计算机业界最后却采纳了后者。凯的 OOP 旨在彻底重新思索我们软件宇宙——今天近乎通用的分隔程序与数据、过程代码与存储信息——的基础之一。假想一下，在新系统中，信息和需要用来解释和操作信息的代码同行同止，就像细胞带着 DNA 封包移动一般。

20 世纪 60 年代在美国空军做一名年轻程序员时，凯观察到这样的计算实现方式——一套借交叉存储数据记录和读取信息所需程序的老式打孔卡片在 Burroughs 大型机之间传输数据的系统。20 世纪 70 年代在施乐 PARC 时，他围绕这种原则设计了 Smalltalk 语言。Smalltalk 的名字是对“那些以印欧神话人物宙斯（Zeus）[①]、欧丁（Odin）[②]、托尔（Thor）[③]等命名的系统几乎无所作为的一种故意而适度的反应。我琢磨，‘Smalltalk’这个名字语不惊人，即便做不了什么像样的事，人家也会一笑置之。”

在一篇关于 Smalltalk 历史的文章中，凯写道：“用计算机术语来说，Smalltalk 是对计算机概念自身的递归。Smalltalk 并没有把‘计算机之类东西’切分为部分弱于整体的小块——比如其他编程语言中常见的数据结构、例程、函数等等——每个 Smalltalk 对象都是计算机全部可能性的递归。所以其语义有点像是用极快的网络连接千万台计算机一般。”结果是“一种更拟生的方案，受保护的普通细胞只通过消息互动，做出人们期望的行为”。

[①]希腊神话中的主神。

[②]北欧神话中爱瑟神族（Aesir）的主神。

[③]北欧神话中的雷神，欧丁神之子。



图 10-4　1981 年 8 月号 Byte 杂志集中介绍 Smalltalk

如果你搞不清楚这种概念，别太难过——有好几代程序员与你为伍呢。令凯沮丧的是，Smalltalk 最根本的创新并未在市场上取得极好进展，在他看来，占据了 20 世纪 90 年代计算机主流的那种面向对象编程根本就是私生子。（“我创造了面向对象的说法，”他说，“还可以告诉你，我看不上 C++。”）用轻便的细胞状对象包裹例程和数据，至今仍未被多数程序员所接受。但对于普通用户而言，如果在从程序之间或系统之间转换时不用再担心丢失数据，或者无须害怕某个特定文档无法在程序或计算机之间转移，那将会是一个诱人的世界。在凯的世界中，对象会“带着所需的全部东西旅行”。

在 PARC 做出突破性创新之后的几十年中，凯看来差不多放弃了劝导那些因循守旧的成年程序员，集中精力于向年轻人展示其概念的强悍之处。最近十年来，除了其他工作外，他致力于设计 Squeak，一种为少儿定制的 Smalltalk 最新开源实现。以凯的逻辑来说，既然我们仍在发掘软件工程的基础，不如让下一代尝尝鲜。

在计算领域中，变化不可避免；所以凯认为，我们设计的系统应该

让我们从变化中学习、并且反过来影响变化。“在 PARC 时，我们的理念就是，既然你永远不会踏入同一条河流，用户界面的第一要素是一种学习环境——可以从不同途径探索，随用户使用该环境的时间进程不断改变。”换言之，用户从计算机处学到新东西后，就能自己修改系统，以期发挥所学。正如道格拉斯·恩格巴特一样，凯希望计算机及其用户能共建一种正向反馈循环，一种启动人类大脑的提靴带机制。



凯是那种特立独行的人，他在各个研究领域取得的成果均受到赞誉，但人们只是择而从之。不过现代软件发展过程中的另一些核心人物也同他一样指出软件行业走错了路。早在 1978 年，Fortran 语言之父约翰·柏柯斯就表达了类似的观点。

柏柯斯认为，编程源自约翰·冯·诺依曼（John Von Neumann）[①]的想法。诺依曼是一位数学家，在 20 世纪 40 年代计算机的黎明时分发明了顺序执行指令的“存储程序”的基本构造。但这些想法后来却变成了紧身衣。“冯·诺伊曼式语言常令我们埋头于地址计算和单词分离计算的尘埃之中。”他写道，“Fortran 及其后续语言也许是自然而然和不可避免地从冯·诺伊曼机的概念中发展而来，可是这些语言却不幸占据我们的思维长达二十年之久。之所以说不幸，是因为我们习以为常，很难再理解和采用新的编程方式，而新方式总有一天会带来伟大许多的智慧与计算的威力。”

我们今天所知的整座软件工程大厦，难道是波坦金村庄（Potemkin-village）[②]的面子房吗？我们是不是该从头来过？

今天，计算机科学家杰隆·拉尼尔也是这种观点的鼓吹者。在 20 世纪 90 年代初虚拟现实技术的短暂狂热中，他满头小辫儿的形象也曾作

[①]约翰·冯·诺伊曼（1903-1957），是 20 世纪最杰出的数学家之一，于 1945 年提出了“程序内存式”计算机的设计思想。这一卓越的思想为电子计算机的逻辑结构设计奠定了基础，已成为计算机设计的基本原则。由于他在计算机逻辑结构设计上的伟大贡献，他被誉为“计算机之父”。

[②]18 世纪时俄国将军格利高里·亚力克山卓维奇·波坦金（Grigori Aleksandrovich Potemkin）为迎接凯瑟琳女皇巡视，将乌克兰及克里米亚沿途村庄粉饰一新，大做面子工程。

为大师级人物在众人心目中留下印痕。拉尼尔说，我们跌入了把计算领域的一切发明都归结为“上帝所为”的思维陷阱中。“看看计算机科学，”在 2003 年一次采访中拉尼尔说，“你会发现文件仿佛就是光子之类自然元素一般。这是种危险的想法。即便你对此无能为力，而且目前你也确实无法脱离文件写软件，但别让自己的大脑受骗却很重要。你得记住，什么是人类的发明，什么不是。”

拉尼尔说，软件领域让人感觉特别像电影《土拨鼠日》——“想法总是雷同，没完没了”——因为我们相信只有现存的计算框架是可行之路。虽然硬件一直在加速，但软件却毫无改进，这是“计算机科学的奇耻大辱”。但程序员们却自满起来，接受了不能令人满意的现状，还视其为恒久不变之事。

在 2003 年的一篇关于“戈耳迪软件（Gordian software）”——像古希腊传说中无法解开的戈耳迪死结[①]一般的软件——难题的文章中，拉尼尔提出了这种对编程现状的批评意见。

如果你观察软件趋势，就会看到摩尔定律的可怖翻版。庞大软件项目的耗费、项目扩充时落后于进度的比率、大项目失败和被放弃的比率，还有由于不可预知的软件问题带来的财务损失都在急剧上升。在所有需要大把花钱的项目里面，正如所料，最常失败的就是软件和医药项目。这并非巧合，因为二者均是我们想要打造成为社群的最复杂的技术。无论如何，软件看起来还是较可原谅一些，因为从直觉上看，虽然它变得越来越复杂，总还没有生物学那么纠缠不清。既然软件是我们自己做的，我们应该能知道如何将其工程化、使之不至于如此混乱。

“计算机科学基础中有些地方原本就不平整，”拉尼尔总结道，对这

[①]小亚细亚 Phrygia 地方国王 Gordius 用绳索绑接牛车的车辕与车轭，宣称谁能解开死结即可称雄亚细亚。亚历山大大帝征战至此，以佩剑斩开死结。

些问题的追踪要回溯到计算领域黎明时期“比照了当时的电子通讯设备”。这些设备都“集中于通过线路”或在后来通过以太传递信号：电报、电话、广播和电视。后来的所有软件系统，从第一个现代机器语言程序到今天广大无垠的互联网，都“模仿了乱七八糟的电报线路”。信号按照发送者与接收者事先约定的协议循这些线路传送。

在传说中，亚历山大大帝拔出佩剑将死结一斩为二，解答了戈耳迪难题。对于乱麻般的思维和意大利面式代码，拉尼尔也提出一种类似的大胆做法。如同艾伦·凯一样，他也是从自然世界中获得灵感。“小改一下程序，就会引起程序所做之事的大变化。假使大自然如此运转，宇宙就会时时崩溃。”而生物系统则能“容错”，维持自我平衡——外人加以干扰后，它总能恢复到原始状态。

拉尼尔建议，放弃从电报时代继承而来的严格协议，造出以躯体连接世界的方式互相关联、并且与人相关联的程序。“我们的神经系统所了解的世界并非基于单点测量，而是基于表面测量。换言之，外部环境不必与我们的躯体预先达成临时语法协议。躯体是与外部世界表面接触的表面。例如，视网膜同时看到多个光点。”何不采用人类用以与现实接口的模式识别原则来制造软件呢？也许该基于可能性而非确定性？它是否该“努力做个好猜谜者，而不是完美的解码器”？

近年来，这些理念帮助机器人领域取得了进步。多年以来，这个领域一直以传统方式痛苦地将外部世界的完美模型逐比特下载到机器中，饱受失败挫折之苦。“当你不再强调协议，注意到表面模式，就进入了一个近乎完美的世界，”拉尼尔写道。“使用协议的话，至少在设计方案的某些方面，你会被拖入全有或全无的高度线性的完美执着中。反之，模式识别假定小错误总是存在，而且并不在意有错误。”



图 10-5 梳着标志性小辫的杰隆·拉尼尔

拉尼尔将这种观念称作“表面互动（phonotropic）软件”。他乐意承认自己关于在本质上“互相守望”的程序的愿景“大有不同、激进、奇异、风险极高”。在表面互动软件中，程序赖以和我们沟通的人机界面和程序之间互相沟通的界面是一样的——“机器和人以同样的方式访问组件”——他承认，乍看这效率太低。但他主张说走这条路能更好地“花摩尔定律的奖金”，用上我们每年新获的增强了计算机威力的芯片额外速度，而现在我们却是在臃肿和注定失败的程序上徒费钱财。

据拉尼尔所言，软件的大问题在于，程序员起步于小程序，并且在小程序的水平上学习原则和实践经验——当这些程序膨胀到今天的大项目一般体量庞巨时，他们发现所有的经验都没用了。“在程序由小变大的时刻，其脆弱性也变得最为明显，而软件工程也就成了薛西弗斯的任务（Sisyphean）[①]。”

在拉尼尔看来，编程这行当饱受某种精神创伤的折磨，那是一种从每个软件开发者在初窥门径时总结到的天真与美德的集体堕落。在计算领域的早期时代，个体创新者也能发明充满英雄光芒的整个软件流派。例如，1963 年为了撰写麻省理工大学博士论文，伊万·萨瑟兰（Ivan Sutherland）[②]写了一套名为 Sketchpad 的小软件（以今天的标准衡量）——独力开创了整个计算机图形领域。“小程序易于编写，而且令人愉悦，”

[①]据古希腊传说，科林斯国王薛西弗斯（Sisyphus）欺骗死神获得永生。神祇惩罚他推巨石上山，至山顶而巨石滚落，如此循环往复，永世不得停歇。所谓薛西弗斯的任务，即指永不终结的无用功。

[②]萨瑟兰因开发三维图形系统 Sketchpad 而被誉为“计算机图形学之父”，后于 1988 年获图灵奖。曾担任美国国防部高级研究计划署（DARPA）信息处理技术局（IPTO）局长。

拉尼尔说。“若可身处能写出革命性小程序的时代，岂不快哉？放弃掉真是痛苦。这出戏一遍又一遍在我们身上重演。在生命及教育过程中，我们体验了计算机科学的整个历史进程。一开始是写小程序——‘Hello World’之类。谁都会获得美妙的体验，开始爱上计算机，事情虽小，但意义非凡。谁能跳过那场初恋呢？它注定了你职业生涯的未来过程。然而，随着你越长越大，一切都变质了。”



图 10-6　萨瑟兰开发的 Sketchpad 支持用光笔绘图

拉尼尔关于“戈耳迪软件”的文章在 Edge.org 网站上引来了暴风骤雨般的批评。Edge.org 是约翰·布罗克曼（John Brockman）为具有未来眼光的科学家们创办的在线沙龙。丹尼尔·德奈特（Daniel Dennett）[①] 指责拉尼尔“寄幻想于一种或许——只是或许——能放大也或许放不大的玩具模型。”拉尼尔没有惊慌失措：他说自己对于软件的批评是一项毕生研究项目的一部分，这项目旨在将计算机改造为让人们可以帮别人实现想法的工具，他把这种互助交换称作“后符号沟通”，将其比作梦想——一种“知觉、醒悟、意识、共享之梦”。

他的动机多少有些理论化和个人化。他认为，从理论上看，解决“规模倍增问题”能帮助我们解决其他问题。“人类面临的基本挑战是理解复

[①]美国著名哲学家。

杂性——在生物学和医学领域如此，在社会学和经济学领域也如此；这全是艺术，全来自我们的科学。无论你沿着哪条智慧路径走下去，总会一再遭遇各种各样的复杂性障碍。所以，理解软件的过程——常被叫做软件的粗糙性——在某种意义上是应付这另一种挑战的主要手段，而且也是最重要而通用的手段。故而我将其看作非常核心的问题，意义深远。如果我们能取得点进展的话。”

从更个人的角度来看，拉尼尔的追寻也是受他对整个计算领域停滞不前的“憎恶”情感所驱动。2004 年在 ACM 会议上的演讲中，他指着笔记本电脑，仿佛它侮辱了自己一般——和二十年前米奇·卡普尔在《软件设计宣言》中描写的一样——高呼：“我再也受不了这些东西的愚蠢！每天就想把这玩意扔到窗外去！真讨厌！我怒火中烧！已经足有三十年了，真是荒谬绝伦。”



和大多数的前卫新概念一样，拉尼尔的表面互动软件是重大突破还是小打小闹，尚不得而知。如今，相对于那一大把急匆匆涌现出来的原型，它还抽象得令人望而却步。然而即便那只是个理论上的大纲，也足以提醒我们软件艺术现今的状态并非石头上的刻痕——其失败乃是人类选择的产物，而非自然规律使然。而且拉尼尔的诊断也划出了思考软件危机永久状态的一条基础分支。我们是否该“面对现实”，假定该领域的进步只能是依靠渐进——对现有模式的适当改进？或者我们应当寻求革命性的新范式——果断斩开戈耳迪死结？我们是该找些小改进来修补软件呢，还是将软件看作拉尼尔笔下那可能“分不清麦金塔计算机和空间加热器”的火星人？

在研究资金大量投入类似量子计算（用量子力学手段替代如今的二

进制电子比特，用原子和光子保存“量子比特”或“qubit”）和基因计算（通过将自然选择原则应用到计算问题的随机产生解决方案池，从而“生长”出而不是写出程序）等试验性领域的时代里，这些问题可不是纯学术的。可关于这些前卫创新是否能造福普通用户，将软件时间的毒害变作我们生命中业已过去的一段，大家还没有达成一致意见。

在篱笆两侧都有智者的声音响起。在《软件极限》一书中，罗伯特·布瑞切尔悲观看待软件悲剧性的极限。他写道，过去半世纪遗留下来的软件成果已然成为我们永远无法摆脱的沉重负担：“一百个、一万个、一百万个计算理论家、实干家和用户创造出来的改进模式困住了我们。再也无法重新开始。”

弗里德里克·布鲁克斯劝诫我们别再寻找银弹：软件复杂性并非一种可移除的品质，而是一种“基本财富”。他还是为逐渐进步留下了空间，主张“竭尽全力、逐步改进”。医药行业自接受微生物理论起就牺牲了魔法药剂的前程，程序员也该放弃包治百病的期望，接纳“为了保持清洁必须坚持不懈地加以照料”的观念。

也有人执着于根本性的改革或革命。麻省理工学院计算机科学家杰拉德·杰·苏斯曼（Gerald Jay Sussman）写道：“计算机科学深陷麻烦之中……问题是结构性的。这不是一种复杂性问题。无法通过某种模块化形式解决。我们需要新理念。我们需要一整套新的工程原则，高效地打造灵活、强固、可扩展和有效率的系统。”



戴维·洛奇·帕纳斯（David Lorge Parnas）发表于 1985 年的一系列文章是软件史上最伟大的文档之一。从其单调的标题《战略防御系统的软件部分（Software Aspects of Strategic Defense Systems）》上看不出来它

将引发一场论战。计算机科学先驱和防御软件工程资深专家帕纳斯当时正任职于一个计算机科学家小组，该小组的任务是为里根总统提议的一套导弹防御计划——主动战略防御系统(Strategic Defense Initiative, SDI，俗称“星球大战”计划) ——提出建议。帕纳斯曾为军方做过大量工作，他也不是什么激烈的反战分子。不过他却于 1985 年从该委员会辞职，宣称 SDI 所需的软件永远无法做出来。

帕纳斯认为，“星球大战”计划决无生机，因为不管是当时的还是能想象到的未来软件工程技术都无法满足程序的需求。帕纳斯极其精确地解释了为什么我们无法造出可用的 SDI 系统，无论是依靠充满低效和缺陷的常规软件开发，还是新语言和新技术，或者软件工程军械库中任何一颗银弹——自动编程、人工智能或者数学化的“程序验证”。

导弹防御系统天生无法在真实的工作条件下测试；在首次“实地运行”时，它就得可靠而高效地工作，保护成百万条生命。帕纳斯写道，理解编程本性的人永远都不会信任这样一套系统。“有位教师告诉我他是如何教编程的，他用最简单的方式解释了今天大多数项目中用到的编程方法。‘像计算机一样思考，’他说。他指示学生考虑计算机一开始会做什么，然后写下来。然后他们再思考计算机接下来会做什么，以此类推，直至描述出计算机要做的最后一件事。实际上，别人就是这么教我编程的。”帕纳斯说，这种方式在小规模的简单程序上工作良好：“我们觉得这管用，是因为它对我们写的第一个程序管用。”然后我们引入复杂的循环、分支和变量。较大的程序关乎多个并发进程。“我们要记住的东西越来越多……最后，我们犯错了。”

如果说“按计算机的执行顺序想事”是程序员的工作方式的话，那么最终这也超出了他们的能力范围。我们是如何做出可运行的软件的呢？“编程是一种试错功夫，”帕纳斯写道，“人们在写程序时，从不指望一次就写对。”他们会犯错，然后再改正。测试和修正。如是反复。

写程序。这句话值得驻足思量。**写**程序。不谈软件领域对架构与架桥等等类推的痴迷以及其间充斥的各种来自生物学或物理学的模型，今天的编程行为仍是一种写作行为——字接着字、词接着词、行接着行。让程序员通过操作屏幕上的图标和图形来做软件的工具有着很长的历史，而且也时有成功（20 世纪 90 年代，Visual Basic 作为其中的杰出代表就广获采用）。不过这些通常都只是覆盖在基于文本的旧代码上的快捷层罢了，要修复真正困难的问题，程序员迟早得深入到代码当中去。

写程序。

比尔·乔伊是同时代的领袖级程序员之一；他编写了 UNIX 伯克利自由版本相当部分的代码，为早期互联网设计了一些关键基础部件，还帮助创造了 Java。离开自己共同创办的太阳微系统公司（Sun Microsystems）后，他原打算写本书。但 2003 年他又告诉《纽约时报》，要把这个项目搁置一旁。这跟撰写让编译器翻译、让计算机执行的东西是两码事；写给别人看还是太难了。“写代码时，”他说，“计算机会告诉你它是否理解你写的东西。写文章就难多了。想要别人能明白，的确很难。”

编程是一种创造性写作吗？这种想法乍看貌似奇谈：有关复杂数学和符号逻辑的学科看来都不会和诗歌及自述摆在一起。不过，Lisp 和面向对象编程老手、如今在 Sun 公司荣任卓越工程师的理查德·戈布瑞尔（Richard Gabriel）认为，编程领域可以从写作世界中学到很多东西。“我的看法是我们应当以训练诗人和艺术家等创意人群的方式来训练程序员。人家会说，‘嗯，听着真不靠谱。’但是，如果有人想要获得诗歌艺术硕士学位，需要做些什么呢？研究诗歌杰作。在软件工程学科里面我们这么做了吗？没有。你不会去看软件中那些杰出的源代码。也不会去看杰出软件的架构。你不会去看人家的设计。你不会去研究杰出软件设计师们的生命历程。所以，你根本就没研究有关自己要创建的东西的文献资料。”



图 10-7 理查德·戈布瑞尔主张编程是一种创作

之所以存在这种情形，原因之一是计算机行业本身缺乏对自身的兴趣。但戈布瑞尔认为，更大的原因是今天多数在用的软件都无法为人所研究；由于商业理由，其代码被锁禁了。（不出所料，戈布瑞尔是开源运动的信徒。）

这就如同所有作家都有自己的“公司”，只有梅尔维尔（Melville）[1]公司的人才能读到《大白鲸（Moby-Dick）》，也只有海明威（Hemingway）公司的人才能读到《太阳照常升起（The Sun Also Rises）》[2]。你能想象得出，在这种情况下居然结得出丰盛的文学成果吗？这种条件既不适合发展文学，也不适合教人写作。我们怎么能指望人家这样学会写程序呢？

戈布瑞尔对编程是一种创造性写作的狂热观念并未停留在纯抽象的层面；20 世纪 90 年代，他脱产学习了 3 年，攻读创作方向的艺术硕士，每天写一首诗。他发现，我们对想成为作家和诗人的学生的要求，比对那些想成为软件开发者的人的要求多：他们得追随导师，他们得在讨论

[1] 赫曼·梅尔维尔（Herman Melville），美国 19 世纪最伟大的作家之一，代表作是 Moby Dick（《大白鲸》）。

[2] 美国 20 世纪大作家亨利·海明威（Henry Hemingway）名作。

会上展示作品并接受他人的批评，他们反复推敲、不断精炼。“我想，对此我们该感到羞愧，”戈布瑞尔说，“我们拿得出手的所谓计算机教育简直就是一出闹剧。”

在其关注的领域里，戈布瑞尔是个激进者——“过去五十年中我们在软件领域所做之事应当全部删除，”他曾这样说过。多数程序员大概都不会对艺术硕士趋之若鹜。多数程序员可能都会像比尔·乔伊那样，感到为一台计算机写东西要比为一群人写东西来得舒坦些。

但在过去十年里，软件行业中发生了一些惊天动地的大事情：程序员们开始个人化、专注、大量地在网站和 **blog** 上写出自己的灵感、沮丧、眼界、技巧、恐惧和梦想。早在互联网发展的最初期，这种举动就在邮件列表和新闻组帖子中出现了。但到了千禧年及网络泡沫破灭之时，才逐渐加速并逼近爆发——那时，许多程序员发现自己整日里无所事事。并非每篇文章都举足轻重，也不是每个程序员都会读这些东西。但它却改变了软件领域——创作出即便不是软件杰作真经，至少也是关于编程日常实践的非正式参考文献。网络本身变作了杰拉德·温伯格在《计算机编程心理学》一书中写到的摆着自动贩卖机的公共屋——一处虽不正式但属必需的地方，编码者们可以在这里分享经验和想法。这也是一个开放式论坛，编码者们可以在其中继续思考、辩论和重新定义他们所从事工作的本质。



是工程还是文学？是科学还是艺术？如何解决编程的双重身份问题，几十年来一直是该领域中许多人的困惑所在——我们并没有显得比北约在佳美善召集大智慧者、请求他们解决软件危机时更加接近答案之所在。或许这难题无解。又或许这根本不是难题，而只是编程作为人类

活动的一种唯一性表现罢了。

你可以在高德纳的生活和工作中找到支持这一结论的明证。高德纳身兼程序员、教师两职，也是在本行中被视作圣经的丛书的作者。在这套丛书中，高德纳探究了编程的数学基础——算法的本质，当我们要求计算机对数据排序或在数组中搜寻或做出其他重复活动时，构成了计算机操作核心活动的那些抽象指令序列。这些书卷代表了高德纳毕生的工作：自 1968 年第一卷问世以来已有三卷出版，两卷正在准备中，还有两卷正在计划编写。这些书与技术书业粗制滥造出来那种《17 秒学会本月流行编程语言（Learn This Month's Flavor of Programming Language in 17 Seconds）》之类手册判若云泥。它们不是写给懦夫看的。

不过，对于那些能跟得上的人，书中对计算领域中数学上极精确、概念上极纯净部分的探索充满了乐趣。书中既有对细节的热情追索，又时时留意不失大局观，融会贯通、手法卓绝。

高德纳给自己的丛书命名为《计算机编程艺术（The Art of Computer Programming）》——而不是《计算机编程科学（The Science of Computer Programming）》。他毕生都在给同事们解释书名，定义所谓编程艺术的含义。他在 1984 年获得图灵奖时发表感言说，“计算机编程是门艺术，”追溯“艺术”一词的用法到中世纪和文艺复兴时期，“当时艺术是指由人类智慧所作之物，相对于源自天然或本能的行为而言。”假使要在人造物和自然物之间划分界限，那么任何与计算相关之事都毫无疑问会落在艺术一边。还有什么能比这些逻辑机器更像是人造的呢？

今天我们惯于区分艺术与科学，而不区分艺术与自然。高德纳欣然承认，计算领域从那条界线的两侧都堕落了。“作为教育工作者和作者，我的首要目标就是帮助别人学会如何编写漂亮的程序……我感觉，在准备编写程序时，可以像是在写诗或编曲……程序员中有聪慧者，有敏锐者，有光芒四射者。我认为，有可能写出**宏伟**、**华贵**、真正**超卓**的程序！”

高德纳像弗里德里克·布鲁克斯一样风格严谨（他也爱好管风琴）。他以如蒙神召般的精纯之心，倾五十余年之力编写和出版《计算机编程艺术》丛书。他持续修订文字，以期不致陈旧，还颁出著名的 2.56 美元赏格（“按 16 进制计的一块钱美金”），指出书中一处错误者即可领取。他在斯坦福大学度过了整个职业生涯，目前是该校荣誉退休教授，编书之余偶尔开设讲座。他曾离开象牙塔进入编程实践世界，将《计算机编程艺术》丛书的编写工作搁置了近十年之久。

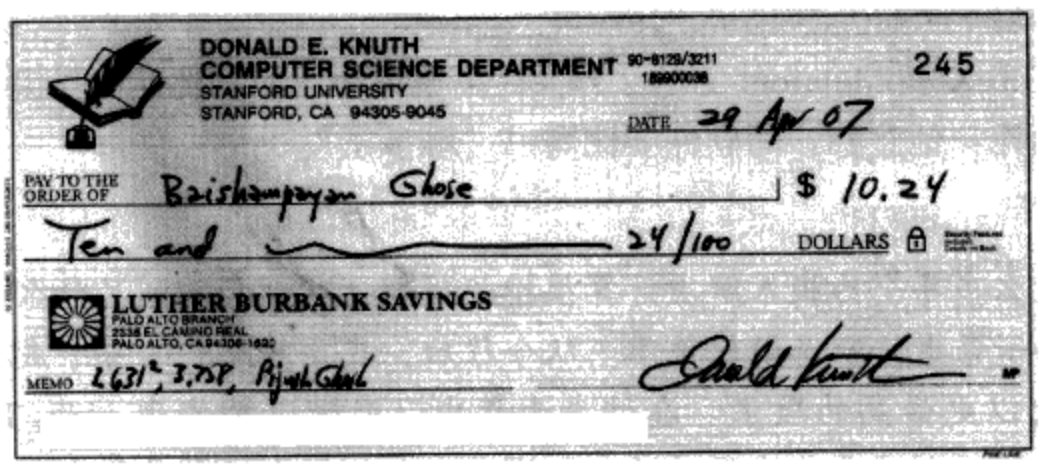

DONALD E. KNUTH
COMPUTER SCIENCE DEPARTMENT
STANFORD UNIVERSITY
STANFORD, CA 94305-9045
90-6129/3211
245
DATE 29 Apr 07
PAY TO THE ORDER OF Bzishampayan Ghose
$ 10.24
Ten and 24/100 DOLLARS
LUTHER BURBANK SAVINGS
PALO ALTO BRANCH
2538 EL CAMINO REAL
PALO ALTO, CA 94306-1692
MEMO 2631², 3.75², Rijndael
Donald Knuth

图 10-8　高德纳为《计算机编程艺术》中 4 处错误颁出的 10.24 美元奖金

据高德纳说，那是在 20 世纪 70 年代，他发现那些新出的技术书——数学和科学课本、需要表现复杂的符号性概念的图书——看起来很糟糕。出版业正经历从“热铅字”时代到“冷铅字”时代、从重金属小方块到软件排版的巨大转变。大笔资金投入到新的基础设备上，而对科学家和数学家的需求则不在优先考虑之列。因为“我不能编写看起来不舒服的书”，高德纳打算为撰写和出版看起来漂亮的技术书编写一套软件工具。

他原打算花一年时间。结果耗时近十年。

高德纳获得了软件时间的第一手经验。最后，他还是写出了程序——TeX 排版系统和 Metafont 字体设计程序——后来都成为相关领域的全球标准。这两套系统可自由获取，经过众多专注于斯且技术高超的用户年

复一年的逐步改进和调试，它们已经成为开放源代码方式的先驱案例。你今天读到的大多数带有复杂科学或数学符号的图书，大概都是用高德纳工具撰写和排版而成的。



高德纳在阅读用 TeX 排版的杂志

高德纳暂别了理论的空中楼阁、在实践的混乱领地上游历多年，这让他对编程战友们面临的阻碍尊敬有加。对于数学家同行们热衷于找出能证明程序“正确性”的数学证据的想法，他总持有疑问。他给某位同事的信中的一句俏皮话后来成了名言：“小心上列代码中的缺陷；我只证明了代码正确性，还没试运行过呢。”

在 TeX 和 Metafont 上的工作，让高德纳得出了与比尔·乔伊正好相反的结论：他宣称，写软件要比写书“难多了”。

这么多年全力投入解决计算机排版难题，从中我学到了什么呢？最重要的一条也许就是**做软件难**。从今往后，我要对见到的每个成功软件工具致以更崇高的敬意。在过去十年中，我惊奇地发现，为 TeX 和 Metafont 写程序要比我做过的任何事（比如证明定理或是写书）都难得多。制作好软件所需的精确性要比做其他事高许多，而且也比其他智慧型任务需要更长的关注期。

这些言语来自高德纳常年开设的系列讲座《理论与实践（Theory and Practice）》；与之对应的幻灯片上写着用他自己的工具创建的描述字体——“软件”二字用了圆角的小写字母 software，而“难”字则是方头钝脑的大写字母 HARD。

1986 年，在富兰克林纪念馆（Franklin Institute）举办的首台可编程电子计算机 ENIAC 诞生四十周年庆典上，高德纳详述了他的意图：

做大型计算机程序需要的关注期要比其他智慧型任务长。我看到《时代》杂志上说，ENIAC 实验室买了些帆布床，好让埃克特（J·普莱斯帕·埃克特[J.Presper Eckert]，负责监建 ENIAC）和其他人可以工作到夜里。在座睡过帆布床的都知道我的意思……大量技术信息必须保存在某人的脑袋里，一次保存全部信息，放在大脑某处的高速随机记忆体里……大型系统的技术性细节数量是让写程序比写书来得困难的原因之一。另一原因是编程要求特别高的精确度。

作为一位穷十年之功对付——并**解决了**——极困难的软件实践问题的地标式作者，高德纳大概要比其他任何还活着的人都有资格比较写书和写代码的困难程度。他的结论是对那些幻想软件可以更快、更容易地制造出来的人的一种温和的警告。但他和许多同事一样，骨子里是乐天派。他认为在 TeX 和 Metafont 上投入的岁月物有所值，不只是因为它们做出的东西，还因为从中学到的东西。他甚至认为，耗去的时间在他的巨著撰写过程中补回来了，因为新工具提高了自己的工作效率。

《计算机编程艺术》向程序员展示了如何设计和测试高效的算法。写出在转瞬间就能对信息排序的代码看似小事一桩，但如果计算机无数次地重复这个算法以完成同一任务，积少就能成多。在这个层面上做渐进微调，需要有与在程序设计和调试层面逐渐改进所要求的同样信念；你得坚信每个小改动都对最终价值有贡献。

丹麦籍诗人/科学家/设计师皮耶特·海因（Piet Hein）的俳句式诗歌精炼了高德纳的信念，这首诗就挂在他家门口通道的墙上：

通往智慧之路？——嗯，一马平川

几句话就能说清：

错

又错
又再错
但更少
更少了
更少了。



这里还要提及高德纳对编程领域做出的另一贡献。尽管他最出名的是在算法上做出的非凡工作，但他还竭力鼓吹一种他名之为“字面编程（literate programming）”的概念。字面编程旨在解决祖尔·索伯斯基所谓的“读代码难于写代码”的问题。鉴于其他程序员迟早都会读到，高德纳建议应强调编写人类可理解的代码，而这样的代码也应有较好的架构。

不考虑我们的主要任务是指导计算机去做什么，集中于向人类解释我们要计算机做什么。字面编程者可以被看做是评论家，他主要关注文笔的优美呈现。这样一位作者，手握词典，仔细选定变量名，解释每个变量的含义。他尽力写出可理解的程序，其概念以最方便人类理解的顺序引入，使用了相辅相成的正式和非正式方法。

在某种意义上，字面编程认为，如果你能完整和清晰地给文档写代码，则代码本身就变得微不足道了。多数程序员太过求快。他们认为文档令人分心。

而且，他们还有注释可用。

注释是代码行之间的英文大白话。每种编程语言都有程序员可用来

标记注释的符号，这些注释旨在帮助其他程序员理解代码；这些符号告诉计算机："这不是给你写的！别关心这些幕后密语！"

有良好注释的代码是优秀编程实践的特点；它说明你关注自己在做的事，而且也照顾到那些跟进修正代码缺陷的人。注释也是一种程序员之间沟通的内部渠道，偶尔甚至还会成为竞技场或排遣无聊的出口。

挪威程序员维达·霍伦（Vidar Holen）精心维护着一个名为"Linux内核詈言（Linux kernel swear words）"的网页。在网页上，他列出了Linux源代码注释中的脏话出现率。其中一幅图表显示脏话的绝对出现率稳步上升；而另一幅图标则显示**每行代码**对应的骂人话数量稳步下降。Linux里面的脏话是多了还是少了呢？霍伦并有没解释这些统计数据。

2004年，当MyDoom、Bagle、Netsky等病毒开始在互联网上蔓延、大肆传染Windows计算机时，有研究者发现，病毒作者们粗鲁地在其他病毒的代码注释中乱加一气："别搞砸我们的生意，你是想开战吗？""Skynet AntiVirus – Bagle –你没救了！！！"这大概就是你能从高德纳处得到的东西了：不妨叫它做字面编程。

注释有时不仅是说明性的注解，还是情绪宣泄的阀门。尽管几十年来自动编程和软件工程的各种概念一直在演进，但做软件还是件痛苦的事。郁闷的程序员有时只想说"fuck"[①]。

程序员在商业代码中这么做，通常是因为他们假设这些话不会外泄，因为公司之外的人永远不会看到源代码。然而我们不时也能窥其一斑。2004年，Windows 2000某个版本的部分源代码泄露到了互联网上。兴奋的程序员们精读了全部文本。他们惊奇地发现，微软程序员们在代码中骂自己、骂工具、骂同事、骂自己的产品。

// around the fucking peice of shit compiler we pass the last param as

[①]由于该词的本义是粗口，反考虑到用原文更可体现情绪，故不做翻译。

an void[1]

// we are such morons. Wiz97 underwent a redesign between IE4 and IE5[2]

//We have to do this only because Exchange is a moron. [3]

// We are morons. We changed the IDeskTray interface between IE4[4]

// should be fixed in the apps themselves. Morons![5]

研究 Windows 2000 源代码的人发现代码中有 4000 多次用到“hacks”一词。为了引起别人的注意，某位程序员在代码中插入以下警告：

[***FOLLOWING NEEDS TO BE COMPOSITED AS A SINGLE-SPACED UNIT[6]

*!!
*!!
*!!!IF YOU CHANGE TABS TO SPACES, YOU WILL BE KILLED[7]!!!!!!!
*!!!!!!!!!!!DOING SO FUCKS THE BUILD PROCESS[8]!!!!!!!!!!!!!!!
*!!
*!!

自北约最好、最辉煌的软件危机解决大会以来，尽管在大约 40 年里发生了那么多改变，今天的程序员还是工作在一个只要把制表符改成空格符就能让他们彻底完蛋的世界里。（不只是在微软；Chandler 用的 Python 也是如此。）他们在注释中发泄的怒气也许不能称之为艺术。但就他们每天仍要应付问题的令人发狂的本质而言，谁又能妒慕他们的自由发泄呢？

[1]向那狗屎编译器传入void作为最后一个参数

[2]我们真是弱智。Wiz97 在 IE4 和 IE5 之间经过了重新设计。

[3]必须这么干，因为 Exchange 太白痴。

[4]我们真是弱智。在 IE4 处修改了 IDeskTray 接口。

[5]应该在应用中修正。白痴！

[6]下列代码需要编写为单倍行距单元

[7]如果你把制表符改成空格，就会被干掉

[8]这样做会弄死构建过程

第 11 章　通往狗食版之路

[2004 年 11 月～2005 年 11 月]

要想在大型软件项目中保持高效，你得敬忠于它。你要将它印在脑海之中。我在做大项目的时候，常常睡觉也梦到代码。

——杰隆·拉尼尔

我们在两章之前离开了 OSAF 开发组，当时是 2004 年秋天，他们已经完成 Chandler 0.4 版，正开始做 0.5 版。这回，他们也给自己设定了目标：Chandler 0.5 将是一个“狗食版日历”——至少他们自己可以用的共享日历。但 Chandler 一直被设想为一种完整的 PIM 软件、一套管理电子邮件、任务和日历的程序，而且其设计方案也要比单薄的日历程序覆盖范围来得大。设计出建造强固、可工作的日历程序，并且为程序功能扩展预留空间，这个任务落在了 Chandler 设计师尹咪咪的肩上。

尹早就发现，在单个窗口里面填上 Chandler 的全部功能和想法并非易事，而消除隔离不同类型数据的地窖的目标更给设计增加了难度。她相信，最终 Chandler 用户会依赖于她提出的仪表板视图——在屏幕上按

时间顺序排列条目——作为日常信息管理的神经中枢。但仪表板功能的开发暂时搁置了；程序员们得集中力量在日历功能上。所以尹就设计了一套 Chandler 界面的临时方案——她希望这套方案不仅对当前以日历为中心的程序管用，而且还能在电子邮件、任务管理等功能加入之后仍可运作。

一眼看去，Chandler 的三栏式结构和其他无数种程序少有区别——包括 Microsoft Outlook 在内。屏幕中部是列出条目的概览视图，然后是细节视图，显示概览视图中被选中条目的详细信息。左侧放置了边栏，用户在这里组织条目集合、选择查看其中某类。Chandler 将包括不同类别的集合：程序自动提供的“开箱即用”型集合，比如电子邮件、日历和垃圾箱；用户自定义的“特别集合”，包括用户自行创建和定义的条目组；还有共享集合——其他人和你共享的条目。边栏不仅需要考虑所有这些类别的集合，还得适应 Chandler 的无地窖结构，在这种结构中，条目可以保存到多个集合里，用户给条目打戳记、将条目从一种类别转换为另一种类别。

尹提议的方案在边栏顶部增加四个按钮：全部、邮件、任务和事件。这个简单的按钮条（最初被称作类别过滤器，后来又取名为应用工具条）将作为一种透镜或过滤器，在你从边栏列表中选择特定集合时，用它来控制在概览视图中显示哪种类别的条目。这套方案力求全面，虽然强大但也令人迷惑：例如，不单是按钮条上有表示过滤任意类型条目的“全部”过滤器，边栏列表上也有“全部”集合，包括资料库中的所有条目。有些使用场景代表了常见的情况：例如当你正在查看“全部”集合，并且单击“事件”按钮，就可以看到“全部事件”——的确，只是个大日历——而概览视图也会切换到日历格式。还有一些使用场景不太常见，反映出 Chandler 对无地窖模式的热望：比如，如果你创建一个包括所有与自家重新装修项目有关的条目的集合，那么单击“全部”按钮就会展

示整个集合，但“任务”和“事件”将会限制这类条目的显示。由于 Chandler 的打戳特性能让你把任务条目变成事件条目，反之亦然，所以有些条目可能会同时在“任务”和“事件”中出现。

尹的计划是在 OSAF 设计组中经过争论和不断修改酝酿成熟的。但当应用组的开发者们审阅这份方案时，他们却大吃一惊、不知所措。其中一些人——特别是约翰·安德森，他正编写边栏代码——对尹的设计方案实在不感冒。一开始，他们发现这套方案晦涩难明；经过几个星期的开会讨论，慢慢涉及细节部分，他们认为这套方案不切实际。

尹的出发点是考虑用户想要如何管理自己的工作流；而开发者们则设想用户对于程序功能建立的“思维模型”。设计师认为，应用工具栏上的不同按钮对应着不同的用户行为模式。开发者们认为，既然她想要提供的其实是同类集合的一组**视图**，那何不照直说呢？尹将应用工具栏看作妥协的底线，是最初设计方案唯一剩下的残片，而原方案中包括了许多志存高远的组件和功能；但开发者们发现她的草图野心太大，满是需要开发人员特制的形状和特性，在他们做好的 wxWidgets 工具集中根本没有类似的东西可用。

程序员们并没有结成反对设计方案的统一阵线；有些人从哲学上反对，有些人认为这套方案还没想清楚，还有些人觉得也许可以一试。但他们明显感到不舒服，有关这个主题的会议也陷入了兜圈子的困境。在一次充满了细节争论的长会结束后，约翰·安德森说他在找“方案中更多具体的设计”。

凯蒂·帕兰特一贯镇定自若，此刻也反驳道：“说到‘更具体的设计’，你得具体说明要什么东西更具体！”

到了 2004 年 11 月初，整个讨论陷入僵局。

有个周四——刚好是总统大选前一天——尹走到唐·登曼桌旁，说要跟他讨论一下。他们聊了一个多小时，想要填平设计师和程序员之间

的鸿沟。登曼回家后看了大选投票结果，然后关掉电视（“真不爽，”他告诉我，“我在俄亥俄州长大的[1]”），给 OSAF 开发邮件列表写了封信，大概列出他和尹达成的协调意见。结论是将类别按钮和集合列表的交集看做一种二维表格；虽然用户看不到表格线，但如果这样来看设计方案，由此建立思维模型，开发者就能比较容易地接受它。

[1]俄亥俄州向来是美国总统大选的关键战场，2004 年大选中，小布什在俄亥俄州获胜，该州民主党人指控他作弊。

那天晚上 12:34 分登曼发出邮件。虽然它还未能解决问题，但也为打造 Chandler 0.5 版的可工作边栏扫清了道路。



“设计师都该花点时间学编程，”约翰·安德森在一次吃午饭时宣称，“这样他们就不会老是提出这么难实现的建议了。”

“那样他们也就提不出什么好点子了，”丽萨·杜索特答道，“他们会固步自封。一旦你知道怎么编程，你就会想，做那样的东西太难了。”

“嗯，”安德森说，“我还是会有点子。但我会用不同的方式来考虑！”



2004 年 10 月，泰德·梁在 blog 上贴出后来在开源程序员中广为流传的一篇文章，文章说 OSAF 正在寻找一位“Python 黑客好手”，请他帮忙重新架构和打磨 Chandler 的“内容模型”——程序用来访问存储数据的一种定义。谁都知道，在这上面的改进将是加速 Chandler 开发的一大步——帮助程序运行得更快，也帮助开发者写得更快。在 dirtsimple.org 网站——网页顶端开玩笑地写着“使易者难，使复杂者成为不可能（making simple things hard, and complex things impossible）”——上写 blog

的 Python 程序员菲利普·J·艾比（Phillip J. Eby）在 11 月份做出了回应。艾比花了点时间研究 Chandler 的代码，然后在博客上贴出一篇长文，题为《Python 不是 Java（Python is Not Java）》。

我最近在看一套基于 wxPython 的 GUI 应用程序的代码……代码的作者是一些 Java 开发者，相对而言他们是 Python 新手，而且代码也受性能问题所烦扰（比如启动就要花 30 秒）。检查代码时，我发现他们做了好些在 Java 里面正确、但在 Python 里面却严重错误的事情。这并不是因为“Python 比 Java 慢”，而是因为在 Python 中还有更简单的方法能够达到同样目的，而这在 Java 中绝无可能。

所以，这些可怜的家伙做了超出所需的工作，写下超出所需的代码，而这些代码的性能要比符合 Python 惯例的代码慢许多，这真是悲哀。

艾比开列了许多技术建议，循这些建议写代码，才会表现得像是真正的 Python 老手而非新入门者。艾比总结道：

这只是从 Java 移植到 Python 时遭遇的冰山一角……基本上，如果你用 Java 已有一段时间、刚开始用 Python，就千万不要相信自己的本能。你的本能已调至 Java 一方，没指向 Python。先退一步，最重要的是别写那么多代码。

要做到这些，就得对 Python 提出更多要求。假装 Python 是一支魔棒，只要你心之所至，无须自己动一根指头，它就能奇迹般地做到。问一句“Python 已经如何解决了我的问题？”以及“什么 Python 语言特性最像我的问题？”，你绝对会惊奇地发现，你需要的东西往往已经以某种形式存在于斯。



菲利普·J·艾比

尽管艾比没指明是 Chandler，但谁都知道他说的是谁。他的观点不可置疑——不管 Chandler 开发者们写了多少代码，他们都还是 Python 新手，就是没能用尽 Python 语言特性的优点，而且有时还用错了。

PJE（艾比的惯用签名）公然揭 OSAF 团队的短，委实**不太厚道**——而且当时他正接受面试、准备加入 OSAF。其他组织也许会对这种行为予以打击。不过在 OSAF，卡普尔一贯鼓励接受批评："如果我们还算个什么的话，"他会说，"我们就是一个知道如何学习的组织。"

于是艾比于 2005 年 1 月加入团队，在位于佛罗里达棕榈滩（Palm Beach）的家中远程工作，即刻开始起草一份建议书，并预备进行大量的代码清理工作。有些工作很重要，但基本上属于清扫性质，旨在"削平树冠"——移除源代码中文件目录的繁复层次，建立一种更简单、更易于把握的结构。

艾比关于重新考虑 Chandler 代码基础结构的想法在概念上更加大胆：他设计出一种名为 Spike 的"可能性推测原型"，旨在"尽量减少人们为了做某种有用的事而需要理解的事物的数量。"Spike 把 Chandler 的底层数据概念重新组织为经仔细定义和整齐分隔开的五个层次。这给了 OSAF 开发者们一个干净的新模型，但也需要清理 Chandler 的许多部分、重起炉灶——在投入了这么多时间和精力之后，这可不是什么醉人美景。帕兰特和杜索特不肯全盘接纳 Spike，只同意从中撷取可以整合到 Chandler、且不会影响其虽然缓慢但却稳定的进程的好想法。

艾比再次在 blog 上报告自己的工作。他阐述了创造 Spike 的过程：从极限编程中借用一种叫做"测试驱动开发"的方法，使用这种方法的程序员在编写功能代码**之前**先编写估量方法成功性的测试代码——这样，他们在每个工作阶段都能确保代码能够运行、执行预定任务。艾比感到，Spike 成果显著："是一种美好之物，一串连锁特性的耀眼宝石。"

艾比在附言中写道，上述自评看来有些自大，其实他的本意是赞扬测试驱动方法，而非自吹自擂。



在 0.4 版开发过程中，OSAF 采纳了新的 Chandler 共享功能实现手段——放弃以前的端到端方式，采用丽萨·杜索特提议的 WebDAV 方式。摩根·萨奇已开始编写将 Chander 条目送到 WebDAV 服务器然后再取回的代码。但服务器到底是什么样？当时还没多少成熟、广泛部署的开源 WebDAV 服务器，而且都没实现面向日历的新 CalDAV 标准。此外，OSAF 开发者们还希望服务器能有一些额外的功能，因为他们想要存储一些特殊的“Chandler 式信息”，Chandler 将会向服务器传递、并从服务器取回这些信息。

杜索特逐步说服大家在 OSAF 的保护伞下开设单独的服务器项目。它将为 Chandler 服务，但与 Chandler 项目分隔开。2005 年初，服务器项目——依拉布拉多狮子狗 Chandler 的年长同胞之名，命名为 Cosmo——开始运转。该项目投入不足，大体只有布莱恩·莫斯利（Brian Moseley）一位程序员负责。不过，由于它没有历史包袱——而且与 Chandler 的交互将遵循 WebDAV 标准——所以也不必唯 Chandler 马首是瞻。例如，莫斯利就决定选用 Java，不用 Python。

Cosmo 很快就有了专属邮件列表，杜索特开始重新组织 OSAF 的 wiki 页，使之能容纳两个单独项目。莫斯利喜欢在家工作，很少在 OSAF 会议上露面。如安蒂·维达预言的那样，搭建服务器的决定不但让整个 Chandler 工作量变大，而且还让 OSAF 组织开始出现分支。如果 Cosmo 遇到麻烦，就会引发恐慌。但服务器项目进展顺利，没有遭遇明显的危机或延误——与 Chandler 面临的路障恰成鲜明比照。

当然，Cosmo 只用了一名程序员，这很有帮助；不必在沟通上花太多成本。不过编写服务器软件与编写用户端软件相比也确有一些优势。服务器是一种几乎只与其他程序和机器交互的程序；它很少需要和人类直接沟通。在需要这样做时——例如，初次配置时或遇到问题时——与之交谈的人往往是位专家、一位通晓服务器语言的系统管理员或程序员。

在 Chandler 团队努力将人类思维过程和工作习惯与软件系统和界面协调起来时，遇到了吐着毒信的大蛇，而 Cosmo 就不会遭此劫难。莫斯利和杜索特面对的是相对更可管理的问题集——比如，新用户如何设立账户？或者，处于不同时区的用户互相共享日历时如何衔接时间？继 Chandler 变化不定的目标和不断拉长的日程之后，Cosmo 仔细定义的边界成为了一种安慰。

01010101010101011010110110

OSAF 领导层有意在 Chandler 0.5 中缩减他们的野心，瞄向较低的目标。他们暂时忘记了组织整个数据宇宙、让外部程序员能够自由扩展程序的承诺。他们告诉团队成员，Agenda 之魂固然美妙，但眼前我们是不是就该只做个能工作的日历程序呢？

但是到了 2004 年 11 月，进入 0.5 版开发之时，就连这并不远大的目标也难以企及。

只是让应用程序实现其功能也难以做到。他们取得了一些进展：约翰·安德森和唐·登曼根据一套新的“构建块树”设计方案修改了 CPIA 框架，他们希望这套方案能让未来的修改更为快速、容易。戴维·萨鲁维终于差不多修正了自安德森初次将其登记为缺陷以来一直困扰 Chandler 的闪烁问题。有了 WebDAV 的帮助，共享成真——而且，按照一套简单的新“共享邀请”方式，Chandler 用户只要用电子邮件给其他

用户发送一张“票据”，接收者就能得到访问共享日历的授权。迷你日历导航器（多数日历程序中本月、上月、下月日历的缩微显示版本）现在工作正常，而且看起来似模似样。安蒂·维达和泰德·梁通过大量中小规模的调优，提高了资料库的运行速度。

不过这些进展并不太有助于实现卡普尔想要开始使用的狗食版软件。与此同时，新进成员的加入也拖慢了进度。在登曼准备离职期间，OSAF 招进了布莱恩·斯特恩斯（Bryan Stearns），他 20 年前曾与登曼在早期麦金塔团队共事。斯特恩斯接手了细节视图的开发。还有短暂脱离编程工作承担教职的网景公司资深开发人员阿列克·福莱特，以及曾在史蒂夫·乔布斯的 NeXT 公司长期任职、最近在苹果公司负责电子邮件程序开发的格兰特·贝列（Grant Baillie），也在 1 月份加入了 OSAF。长久而言，新面孔的加入能增强力量，但短期内他们得花些时间研究 Chandler。布鲁克斯法则屡试不爽。

Chandler 现在有 150 万行代码，其中多数需要与 wxWidgets 和 Twisted 等其他项目整合工作。OSAF 开发者们编写了 130,000 行 Chandler 专有的 Python 代码。如 OSAF 一位暑期实习生所言，进入这个项目就像是到了一座新城市、拼命想找到自己的方向。

2005 年 3 月 30 日，0.5 版发布一天后，我问卡普尔，项目进行了这么长时间，而且还有很长的路要走，他是否会感到气馁。“我知道，我们会做出某种伟大的东西，”他说，“只是不知道还要经过多少波折。”

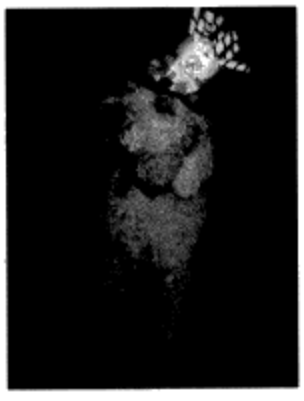


自 2003 年起，每年 3 月，Python 的拥趸们都会聚集到华盛顿特区，参加 PyCon[1]技术大会——开源程序员们理想的交流场合。除了其他奇客式课程，PyCon 还开设“冲刺（sprints）”实验课——短期作坊式课程，

[1]从 2006 年开始在德克萨斯州召开。

参加课程的程序员着手制作功能较少的项目，看看自己在两、三天内能做到什么程度。

2005 年，泰德·梁和另外几位 OSAF 开发者参加了 PyCon 大会，帮助开设一期 Chandler“冲刺”课。他们和三位志愿者组成的团队一起为 Chandler 造出了两个新扩展——为程序添加新功能的代码包。其中一个可以让 Chandler 连接到 Flickr，那个越来越流行的相片共享网站服务；另外一个将 Chandler 挂接到 Del.icio.us——该网站允许用户贴出并共享网页书签，并给这些书签打上描述性标记。使用这些新代码包，用户就能导入和导出 Flickr 相片或者 Del.icio.us 书签，然后在 Chandler 里面管理这些相片和书签。林智超也在资助 OSAF 的大学协会召开的一个技术会议上开设了类似的“冲刺”课；在那期课程中，制作出了能从 Amazon.com 导入公共“愿望列表”的代码包。



3 岁的 Cosmo

只要把这些内容拉进 Chandler，就能以 Chandler 的方式处理它们，重新组织、打戳记、改变其分类。PyCon 大会开完后，阿列克·福莱特向 OSAF 的同事们演示了这些代码包。他用 Flickr 代码批量导入卡普尔照的相片——都是他爱犬的快照。他挑出一张标记为“3 岁的 Cosmo”的相片——一张小狗戴着王冠的生日肖像——然后戳记为那天的事件。他切换到日历视图，在日历上显示出那张相片。

Chandler 的每个扩展，就其本身而言算不了什么，但如同摩根·萨奇的相册程序做到的那样，每个扩展都给 Chandler 最初的承诺注入了生命力。日历是 OSAF 开发者们该全力做的——但就是这些在功能扩展上的小小演习，这些数据炼金术的试验，让他们的热情继续燃烧，在经历如此之多的延误和挫折之后，还能勉力守护 Agenda 之魂。当卡普尔开始向业界的朋友们演示 Chandler 时——Chandler 0.5 版完成后他终于这么做了——正是这些扩展，以及它们几乎不费吹灰之力就把貌似截然不同的信息拼到一起的能力，赢得了一片惊呼之声。

随着时间流逝，某种不寻常的事情在 Chandler 团队中发生了：并非有意设计，可也并不全属巧合，它成了一个主要由妇女管理的开源项目。卡普尔还是“终身仁君”——他半开玩笑地在 Chandler 邮件列表中接受了这个称号，带着向李纳斯·托瓦茨致敬的意思。但凯蒂·帕兰特和丽萨·杜索特管理着那些工程团队，雪拉·莫内掌管产品管理，尹咪咪是首席设计师，Chandler 在软件开发世界中到达了令人难忘的女性领导境界。

做软件的男性通常多于女性，在开源项目中男女比例更不平衡。进入该领域的女性往往做出巨大而影响深远的贡献，但她们的人数却从来都不太多，而且近年来攻读计算机科学专业的女性人数确实也在下降。试图解释这种不平均风险的努力，既引出了陈腔滥调式的言谈，也激起了亵渎圣牛般的高论。但最有思想的研究者们指出，对妇女的社会性偏见远胜于其他因素。在米尔斯女子大学（Mills College）讲授计算机科学、多年来一直研究这个问题的爱伦·斯帕图斯（Ellen Spertus）在 1991 年发表的论文中写道：“尽管谁也不能排除某些先天智力或心理差异导致妇女更（或更不）适合计算机科学的可能性，我发现，对妇女从事此类职业的文化偏见如此之强烈，即便确有天生差异存在，也无法完全解释这种隔离现象。”

如今，每当步入某个软件项目管理会议、看到几张女性面孔，仍然是一种奇特的体验。但凡我问过的 OSAF 成员，谁也不曾在有如此之多妇女做主的软件团队中工作过，而且几乎所有人都觉得这种罕有现象可能是让办公室中人人彬彬有礼的部分原因——粗口对骂、粗暴无礼和斗勇争狠之事相对鲜见。冲突确实存在，但被小心地掩盖了。难道是卡普



菲利普·博苏特

尔为项目定了不同的调子、为妇女搬开了前进的障碍？还是这些才智女性升到高位后创建出一种于己相宜的氛围？

这类鸡与蛋的问题或许没有答案。至少在 OSAF，女性程序员们干得不错。尤其是有一位女性，在过去几个月里被证明是不可缺少的，从而承担了管理职责中的相当部分。帕兰特负责 Chandler 的长期架构事宜，她还是不断新增开发者的应用组的常务经理。卡普尔决定在 0.4 版开发期间为应用组找个直接向帕兰特汇报的经理。不过找人不太容易，直到 2005 年 3 月 0.5 版将近完成之时才有人充任此职——任者乃是行事缜密的法国工程师菲利普·博苏特（Philippe Bossut），他讲话口音浓重，出言略显刻薄。博苏特刚从 Macromedia 离职；和丽萨·杜索特一样，他也曾在微软任职多年。

博苏特甫一接手应用组，即开列出 0.6 版中每位开发者对应的各项任务，还做了个 SWAG——“silly wild-ass guess（盲估）”——估计任务耗时。杜索特曾用过贴纸法，迈克尔·托伊也在数年前要求开发者在 Bugzilla 中填写估计任务完成时间，博苏特的做法貌虽与之离、神实与之合。不过这次在细节上要求更多；用程序员们的话来说，就是“粒度更小”。

“拿到一个大问题，将其切分为可个别理解和解决的小问题，”博苏特解释给我听，“并无魔法。只是得巨细靡遗。”

通过过 4 月中旬的第一轮排查，博苏特发现，每位开发者差不多还得花 40 个工作日才能修正完 0.5 版结束时留下的缺陷。这意味着和之前几乎所有版本一样，0.6 版也得耗费 6 个月以上的时间，而不是卡普尔原本期望的 3 到 4 个月。“我想说，我不介意花 6 个月时间，只要做出我们设想中的 0.6 版就行。”他对经理们说。卡普尔觉得，0.6 版末了最重要的是花点时间在“整装完成”工作上，把用户界面做得清爽利落。那就是发布可用日历程序的最后大障碍——对于技术推广人员们面向的早期

采用者而言并非必要，但至少对于那些可以在 OSAF 内部找到的首批使用者、那些勇敢的灵魂来说，是有必要的。

然后开发者们便投入工作了。他们一口口啃掉了缺陷。他们开始给 Chandler 增加普通但基础的特性——比如日历条目颜色，这样在重叠多个日历时就能知道条目从何而来（例如“家庭”和“工作”，或“我的”和“配偶的”等等）。他们重新编写了表格视图中的表头，这部分一直很难看。他们在迷你日历上加了“忙碌指示条”，显示有哪些日子安排了活动。他们修改了 Chandler 的整个界面，使之具备“国际化”和“本地化”特性，让程序能够按不同语言进行定制，并支持不同的本地习惯，例如每周从星期几开始之类。采用开源手段很容易解决这类问题：IBM 有一套免费库，包括有关语言和本地化的所有必需特性，不管乌拉圭人喜欢从星期天还是星期一开始一周都没问题。它不算完美——实际上，真有一位乌拉圭开发者在 Chandler 邮件列表中反馈了一些错误——但也省下了量比移山的劳力投入。

有些是可管理的任务。不过从 Chandler 的早期开发阶段就遗留下一些不易对付的问题需要解决。其中之一是开发者们所说的“日期/时间挂件”——用户用来设定活动开始和结束时间的工具。在纸面上写出活动日期和时间，方式相对灵活：可以写“5/15 at 3 p.m.”或者“3 o’clock on May 15”或者“15 May@3”，信息都能正常传达。回到 20 世纪 80 年代，卡普尔心爱的 Lotus Agenda 能出色地解析便签上的日期和时间；它甚至能够从“下周一”这样含混的说法中发掘出本意来。谁也没指望 Chandler 能做到——总之现在不行。不过尹咪咪希望至少能灭掉计算机日历的多数恼人特性。她主张 Chandler 应遵循人类表达的方式来构建时间-日期条目，还建议在细节视图中提供一个单独区域，用户可以在这里输入类似于“Jan. 3-5, 2004 from 4-6 pm”的文本。

“那信息可太复杂了，”细节视图的守望者布莱恩·斯特恩斯反对道。

在开发者们看，打造聪明、对人类友好的日期/时间挂件来是另一个野心爆棚的进度杀手。一番唇枪舌战之后，尹的设计主张败在了他们出于实践的反对意见之下。在 0.6 版中，每条 Chandler 日历事件条目的开始和结束期都有单独的日期和时间字段——总共 4 个字段。



假使 Chandler 0.6 要让首批用户用上真可使用的日历，就还要加一个特性：重复事件。有些条目不但想加到日历里，还想让它们按周期重复发生——比如公司部门的每日例会，或者小孩的每周吉他课。印刷日历没有这类服务，但这正是人们希望软件能轻易做到的事。

然而，周期重复特性却给 Chandler 开发者们带来了一些既现实又富有哲学意味的难题。如果想要将某个事件设置为按特定周期发生——比如，下个月的每星期——没问题。但如果不愿意设定终止日期呢？如果想要让事件一直重复、直至自己决定将它从日历上去掉呢？

软件不太善于对付无限性。计算机都有存储容量限制，当程序陷入无限循环——一个没有出口的例程——就会停止回应。多数日历程序都找到了解决不确定重复期限的方法：只要保存重复时间的一个“主”版本，根据其限定规则显示即可。也可以选定未来某一年度，设定重复期限到那个时间点为止。在 OSAF 研讨这个问题的时候，丽萨·杜索特汇报说，在另一个系统当中，不确定重复事件将不断复制自身，到 2038 年就将终止（如同汽车里程表到达最大数字一般，所有基于 Unix 的计算机系统内部时钟也会在 2038 年归零）。“显然，”杜索特在一封电子邮件中写道，“世界终结于 2038 年。就好的一面而言，2038 年后过生日再也不会变老了。”

这些经过验证的解决方案总会留下些首尾。例如，对重复事件其中

某一次做修改，比方给当天的会议添加备注，修改会传递到后续的事件中吗？如果你打算采用单个主事件手段就会这么计划。那么，如果想给某个会议事件的单次活动添加备注呢？尽管许多成熟的日历程序都找到了解决这两难问题的方法，但新来者仍然得再解决一次。

Chandler 的独特设计又带来了更多问题。如果把重复日历事件戳记为任务会怎样——是只给单次事件打戳呢，还是给以后的每次事件都打戳？如果在概略或列表视图而不是以日历格式查看事件，会看到没完没了重复出现的不确定条目吗？用户界面上有问题，后台资料库也有问题。也有办法把问题推后——当用户真的查看某个特定时段时，只生成重复事件在该时段内的次数——但都是权宜之计。格兰特·贝列指出过一种麻烦的边缘状况："如果用户（可能是误击，或者出于好奇想要看看自己 40 岁生日那天是星期几）跳到 2025 年——难道我们要把这 20 年的重复事件都保留到资料库中不成？"

2005 年春夏两季，Chandler-dev 邮件列表里面充斥了如果实现不确定重复事件的讨论——还有针对如何限制其无边界特性的难点的讨论。阿列克·福莱特在帖子中写道："窃以为这是 Chandler 中最难的问题。"



为什么貌似简单的事件重复问题实则如此之难，可以在"重复（recur）"一词上找到点线索。不仅是整个行业能感受到（就像米奇·卡普尔和杰隆·拉尼尔在不同时期的抱怨一般）的在做软件时如电影《土拨鼠日》中循环时间旅行般怪异地重复出现的问题。"重复（recurrence）"也是"递归（recursion）"的另一种表现形式，而递归则正是软件中强大、有时野性难驯的核心之物。

重复者复现，而递归者则以特殊方式复现：往复到自身。它们是吞

食自己的蛇。

以算术而言，“递归”描述了一个调用自身的函数；在计算机科学中，递归表示以自身定义的函数。递归逻辑给出了一种生成复杂、精细输出的简约表达法。它位于许多计算领域最著名、最令人鼓舞的系统及理念的中心处。道格·恩格巴特关于提靴带的概念是一种加速人类进步的递归技术。Lisp——约翰·麦卡锡于 20 世纪 50 年代后期发明的语言，几十年后仍然鼓舞着许多 OSAF 的程序员——就是一种伟大的递归机。艾伦·凯喜欢指着麦卡锡“Lisp 1.5 版手册第 13 页底部的半页代码”，赞扬它是“计算领域的麦克斯韦方程组[①]”——在机器时代的黎明时分以精练、优雅的表达式提炼了软件领域的基础原则，就像詹姆斯·克勒克·麦克斯韦（James Clerk Maxwell）的四个方程式铺陈出电磁的基本运动一样。在凯引述的那一页上，给出了“eval”和“apply”两个函数的定义，麦卡锡从本质上用 Lisp 描述了 Lisp。凯说，“这是整个编程世界中我能覆掌于上的几行。”

你能覆掌于上，但可不总是那么容易一手把握的。递归令人头疼。侯世达（Douglas Hofstadter）[②]的大作《哥德尔[③]、艾舍尔[④]、巴赫[⑤]——集异璧之大成（Gödel, Escher, Bach: An Eternal Golden Braid）》[⑥]可能是对这个概念面向非数学家的最易理解和最可接近的解释了。侯世达将数学领域中神秘的自引现象与艾舍尔（M.C.Escher）艺术作品中循环上升的阶梯，以及巴赫（J.S.Bach）充满趣味的卡农与赋格曲连接起来，给这些现象贴上一张令人难忘的标签：奇异循环。侯世达用了将近 800 页篇幅就这一主题的各种变调逐一条分缕析；哥德尔、艾舍尔和巴赫都重复着自己的主题——奇异地循环着。

在艾舍尔楼梯上的人可以不停上攀、永不停歇。但软件中的递归很危险，因为它必须要有出口。否则，奇异循环就会变作无限循环，而计算机就——使用精确的术语说——漫水了。如一本 Lisp 课本所写：“如果递归函数没有适当的终结条件，就会导致堆栈溢出……终结条件是递归

[①] 英国物理学家 J.C.麦克斯韦在 19 世纪建立的描述电磁场的基本方程组，包括四个方程，不仅分别描述了电场和磁场的行为，也描述了它们之间的关系。

[②] 美国著名学者，计算机科学家，印第安纳大学计算机科学和认知学教授，观念与认知研究中心主持人，哲学、心理学、比较文学、科学史与科学哲学副教授。

[③] 20 世纪最伟大的数理逻辑学家。

[④] 荷兰版画家，错觉图形大师。

[⑤] 德国著名作曲家。

定义中最重要的部分。”

重要之处在于确认每个函数例程都有“终结条件”，而不是无休止的循环，在我们意欲为之时，计算机科学又让我们面对令人烦恼而又无可奈何的事实：我们做不到。即便可以尝试，但不总能绝对确定成功了。

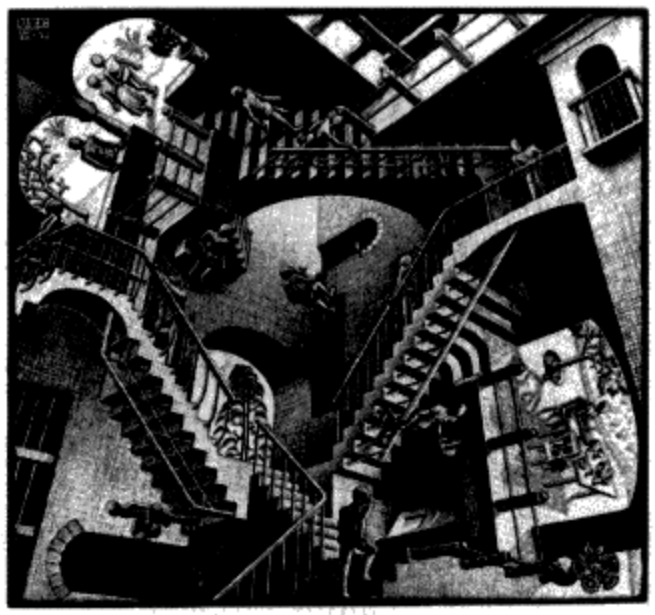


图 11-1　艾舍尔画作中循环上升的阶梯

20 世纪 30 年代，数字世界初露峥嵘之时，英国数学家艾伦·图灵（Alan Turing）定义了后来成为现代计算领域基石的基础抽象，他设计出令人吃惊的证明：他论证了不存在无所不能、包治百病的检测任意给定程序机器初始输出、并确认程序是否“结束”（或遇到“终结条件”）的方法。也就是说，不存在能够证明任意给定程序接收全部可能输出均能结束的通用算法。换言之，你写不出那种确保其他程序无论输入任何数据总能最终结束的程序。

戴维·哈瑞尔（David Harel）在其著作《计算机公司（Computers Ltd.）》[①]中写道，“通过模仿运行程序……并等着看结果的仿真算法来解决问题，这极具诱惑。问题在于执行何时结束，在结束时我们是否能适

[⑥] 本书常缩写为 GEB（数学家哥德尔、艺术家艾舍尔、音乐家巴赫三个名字的字首）。书中通过对哥德尔的数理逻辑、艾舍尔的版画和巴赫的音乐三者的综合阐述，引人入胜地介绍了数理逻辑学、可计算理论、人工智能学、语言学、遗传学、音乐、绘画理论等，是在英语世界中有极高评价的科普著作，曾获普利策文学奖。

[①] 全名 Computers Ltd.: What They Really Can't Do。

时停下，推论答案为‘是’[程序的确结束了]……难点在于判断何时停止等待，说一声‘不’。我们不能在等待许久之后就这么放弃，推断说仿真还没结束，所以永远也不会结束。”回答这个问题需要无穷的时间；无论是人类还是世上最快的计算机，都没那么奢侈的资源可用，所以这问题也不可回答。



图 11-2　计算机先驱艾伦·图灵

谁也没能驳倒图灵这套虽然与直觉不符但却在逻辑上可论证的理论。这套理论人称“停机问题（**halting problem**）”。坐在分隔可计算和不可计算的边界之外，它首次表达的概念后来成为令人惊异的一大套抽象问题，即便是拥有难以置信的速度与威力的计算机也无法解答。

类似停机问题这样的谜题，还有那些令计算机科学家彻夜难眠的不可判定问题，从未阻碍常用软件的编写——我们用这些软件跟踪盈亏、做日历记录或是撰写图书。但它们确实推动了软件进步。关于自动程序“验证器”——一种能够检验其他程序、判断受检程序是否对任意给定输入提供正确输出的程序——的梦想注定只是梦想。这一问题不可解答：程序可能对许多个案做出精确验证，但永远不能绝对确定在所有案例中始终正确。如哈瑞尔所言，这“粉碎了我们对一种能确认计算机按需而行的软件系统的期望”。这种系统不只是难以想象、难以设计、难以实现；就目前我们最高的数学理解水平而言，它根本不可能实现。

停机问题和类似的不可解决的难题都是形式逻辑问题，要求通过符号表示法来定义。在用程序员名之以自然语言的东西来解释它们时，精确性就丢失了，就像现在我做的这样。不过我们还是有所收获。它们也是有用的隐喻。

当 OSAF 开发者们纠缠于如何在 Chandler 中实现重复事件之时，我发现自己正面临自作自受的停机问题。我观察卡普尔及其程序员们的工作将近三年。2003 年 1 月我初次在 OSAF 露面时，他们相信一年后能完成 Chandler——至少是可标记为 1.0 的可用产品。**当然可以**，我那时这么想。我去过那个街区一、两次；在安排写作进度时，我立即对他们的计划产生了怀疑。我想，我要做两年研究——那时 Chandler 也该完成或者关闭，然后我就可以开始写了。

然而我的怀疑还不够充分。Chandler 的进度像是个冗长无聊的故事般延伸。直到第三年，还是看不到“终结条件”。这故事到何时才会自然结束？往坏处想，我能否确定它会有结束之时呢？

做写作计划时，我没考虑到侯世达法则——侯世达用自己名字命名的递归原则：*时间总会比想象中用得多，即便你考虑到侯世达法则亦然。*

这奇异的循环看来定义了软件时间的本质。现在，我作茧自缚。

经过三年之后，Chandler 开始成了某种虽未完成但还算可用的日历程序。然而我没有信心说这项目还要多久才能交付如初所愿的软件。OSAF 的经理们业已计划大概再花两年时间做四个“点版本”，之后才是 1.0 版——不过程序的形状和涉及范围却仍向未来的修订和修正敞开。

似乎自从我开始探访 OSAF 开发者们，他们的目标就一直在大概两年以外；随真实时间流逝，目标也在后退，就像不可触及的天际线。他

们的工作帮我找到了起初想到的关于软件的一些问题的答案。而现在我的故事线索却开始在软件时间的黑洞里渐渐消逝。



就像写软件时遇到的许多问题，如何在 Chandler 中处理重复事件的问题也被证明是一种分而治之的系统化战役：将难题切分为可管理的多个部分，解决其中足够多部分，使之可以运行，最难的部分则留给不可知的将来。负责解决大多数递归难题的开发者杰弗瑞·哈里斯（Jeffrey Harris）最初是项目的远程参与者，在邮件列表中发言，贡献软件补丁包。哈里斯曾在密歇根大学攻读物理学，后在密苏里州加入一个叫做“舞兔生态村（Dancing Rabbit Ecovillage）”的“概念社区（intentional community）”，造房，唱歌，经营软件咨询业务。

2005 年夏天，他移居到旧金山，全职加入 OSAF，成为 OSAF 日益扩张的雇员队伍中的一员，这些人最初都曾远程为项目作贡献，或者做项目志愿者。“代码熊”麦克·泰勒（Mike “Codebear” Taylor）住在费城，为 Chandler 贡献建议和想法已有数年，OSAF 请他负责管理富有技巧的构建工作——管理代码树，监控里程碑版本结构，确保在 Tinderbox 图表变红、构建失败时尽快修复。

这种核心操作角色可以交给总部以外美国大陆另一端的某人来负责，标志着 OSAF 朝向“真正的”开源项目方向又前进了一小步。2005 年，OSAF 开发者们也把更多的技术争辩从会议室和邮件往来中搬到项目公共邮件列表上，好让志愿者和外部人员能平等参与。

但耗时三年编写开源代码和搭建一个开源社区可不是一码事。在 Chandler 0.6 接近完成之时，泰德·梁给米奇·卡普尔递交了一份关于评估 OSAF 在开源方面成败的报告。他发现，当时在 Bugzilla 中记录了大

约 4,400 个缺陷，其中差不多只有一百个来自外部人员。卡普尔及其团队创建了某些成功开源项目所需的前提条件，比如遵循某个开源授权公布源代码，在互联网上公开他们的研究成果和考虑过程。如不透明，就无价值。但是，梁写道："在为可见而透明和为协作而透明之间仍有所差异。"

要实现后者，OSAF 还有很长的路要走。Chandler 集市中的帐篷还不够多。而且开源也不能让 Chandler 免受其他无数野心勃勃的项目所遭遇的时间扭曲之痛。也许艾瑞克·瑞蒙德的《大教堂与集市》错了，李纳斯法则（"眼球足够多，缺陷无处躲"）还是没能超越布鲁克斯法则。或许 OSAF 的透明度还是没能满足瑞蒙德关于成功的集市型开源项目的要求——它必须认可、采纳和奖赏来自外部人员的好主意。

自由软件教父理查德·斯托曼喜欢说："如果有人问我某事何时结束，我总答道，'只要你来帮忙，就会完成得快些。'" OSAF 欢迎志愿者和外部贡献，但 Chandler 宏大的设计蓝图和迟缓的交付步调却让外人难以追随。

在《大教堂与集市》中，瑞蒙德写道，"显然没人能以集市方式从头开始写代码。可以用集市方式测试、除错和改进，但很难以集市模式新创项目……初生的开发者社区需要有可运行和可测试的代码来玩。开始构建社区时，要呈现的还只是一个模糊前景。程序无须工作得特别好。粗糙、多错、未完成、缺文档也行。不可或缺的是说服潜在共同开发者，在可预见的未来，它能演化为某种真正的妙物。"



我和 OSAF 外部的人谈及 Chandler 时，最普遍的批评大概如此：卡普尔没从一开始就在浏览器内部构造 Chandler，他错过了船期。那样做可以节省为不同操作系统打造跨平台软件的额外工作，还有为构造基于

wxWidgets 的复杂图形界面付出的劳动。网页浏览器搞掂一切——它业已运行在多个平台之上，提供了现成的用户界面。到了 2005 年，人们都在说，基于 Web 的软件显然已是未来趋势。Gmail 风格的 AJAX 技巧风行。Firefox 炙手可热。浏览器正当其位。

卡普尔坦然面对批评。首先，他指出，2002 年 Chandler 的架构刚建立时，世界与今不同，如果不能在线工作，基于 Web 的软件就无用武之地（话虽如此，宽带和无线连接在美国到处可用，多数人只在飞机上离线）。最后，基于 Web 的 AJAX 式手段，虽然令人兴奋，但并未提供那种符合 Chandler 瞄向的新颖、灵活设计的媒介。

尽管如此，卡普尔还是在 blog 上承认他已经开始用不同眼光看事物。这位自称“搞应用的老伙计”现在也近乎接受通过 Web 交付、基于浏览器的软件开始取代个人计算机黄金年代基于桌面程序的地位这一现实。从现在起，他宣称，倾向于从新项目一开始就采用网页界面。

2005 年夏天，随着 Chandler 0.6 和 Cosmo 服务器早期版本工作的进行，OSAF 又启动了一个小项目：网页或客户端界面，它让用户可以访问保存在 Cosmo 服务器上的日历。不只是为 Chandler 用户服务，还为其他日历程序的用户服务——包括麦金塔的 iCal 以及支持 CalDAV 新标准的其他日历软件。

卡普尔常说，OSAF 应当致力于创建好日历的原因之一是人们真的需要它。鉴于 2005 年涌现出来提供这样或那样日历服务的初创小公司和软件项目的数量，他说对了。但新的日历服务越多，用户和程序员也越来越面临日历之间不能兼容的问题。这已经是“火车出轨”了，Infoworld 专栏作家扬·乌代尔（Jon Udell）这样写道。软件作家司各特·麦斯（Scott Mace）开设了一个 blog，追踪这个问题；他将 blog 命名为“日历沼泽（Calendar Swamp）”。

OSAF 开发者们期望他们的 CalDAV 标准能把世上这些松散的日历程序绑到一起，帮忙抽干沼泽地。丽萨·杜索特和布莱恩·莫斯利着手招聘两三名开发者，负责搭建新的网页客户端。依据 OSAF 用狗的名字命名的范式，他们将这个项目命名为 Scooby。



原本计划于 2005 年 9 月发布 Chandler 0.6。到了夏末，缺陷仍旧堆积如山，经理们同意将发布期限推迟到 11 月底。这次，他们坚信，一定要做出狗食版来——他们要以某种临时手段发布一套真正可用的程序。而且他们也知道，这将改变他们自己的世界：不再瞎猜下一步怎么走，倾听来自真实用户的建议和抱怨。这些真实用户是在千里之外还是就在会议室里与他们邻座而谈并不重要；重要的是拿到实践反馈。

但在 0.6 逐个实现里程碑时，发生了出乎意料的事。OSAF 的人们开始提早试用 Chandler。有些事赶过了进度。这次，软件时间跑过了真实时间。

2005 年 9 月 23 日，菲利普·博苏特开始用 Chandler 做自己的个人日历管理。他几乎是立即就发现了新缺陷，特别是在重复事件部分。他在 blog 上写到这次体验："我遇到几个致命错误，在平常使用时就小心翼翼。我避免冒险使用重复事件（比方共享或者在群集间移动……）。撇开这些缺陷不提，好消息是：Chandler 正在很快变得可用，我用 iCal 能做到的事，没有用 Chandler 做不到的。"

很快，雪拉·莫内也开始试用 Chandler。她丈夫找了份新工作，正要搬到温哥华去，她自己每个月会在那边工作一个星期。这样的空间距离关系让她得以测试日历共享特性。

到了 10 月底，卡普尔自己也开始试用——把 iCal 事件导入到

Chandler，用 Chandler 做日历管理。还有个障碍：Chandler 的重复事件特性允许以每天、每周、每月的周期重复。但卡普尔的日历上有几个条目是每两周重复一次的。

在 0.6 开发周期的这个时间点上，软件版本被看做已经“特性完备”；不再添加新特性，只集中修复最高优先级的缺陷。不过开发者们稍稍越过了界线：让 Chandler 0.6 支持创建每两个星期发生的重复事件。



停机问题不光是个理论谜题。每个软件项目的参与者在开始寻求工作的终点时都会一头撞到停机问题的真实世界版本上。而每个团队都学到同样的痛苦一课：世上没有能够告知项目何时结束、是否达到目标的简单可靠的规程。

1999 年，艾伦·库珀（Alan Cooper）——在 20 世纪 90 年代创建了大部分 Visual Basic 编程语言的软件开发者，后成为卓越的软件设计鼓吹者——写了一本名为《精神病患管医院（The Inmates Are Running the Asylum）》[①]的凌厉之作，给出了一份软件行业所犯罪行的登记表。库珀在书中写道：“软件开发缺少一个关键元素——理解什么是‘完成’。”

停机问题在软件开发进度上有实际反映——Chandler 开发者们与之对抗达三年之久。当软件产生于对投资者负责的新创公司的坩埚之中时，财务紧缩及资方要求通常是项目的强制“终结条件”。而纯开源项目的进展则只受志愿者投入力量之消长所制约。用李纳斯·托瓦茨的话来说：“软件是免费的，所以也没有仅仅为了达到销售目的就在还未准备好就之前发布的压力。Linux 的每个版本都是在真的完成后才宣告完成，这也是它如此强固的原因。”Chandler 坐在这两种模式不寻常的中点处，没有清晰标示“完成”时刻，甚至没有指出何处可能是终点的路标。

[①]全名 The Inmates Are Running the Asylum: Why High-Tech Products Drive Us Crazy and How to Restore the Sanity，中译本改名为《软件开发的创新之路》。为配合行文风格，此处特直译书名。

但软件开发的停机问题还带来一个结果，一种重存在轻实际的结果。《搞掂》大师戴维·艾伦谈到过困扰那些面对堆积如山开放任务的知识工作者的“恼人焦虑感”。软件开发者总有工要赶；“完成”的定义，即便是对于临时版本或小里程碑，也总是主观武断的。在这一点上，他们的工作更像是作者，不像是修建者。“完成”对于观察者而言并非显见。“完成”是你需要自作主张的决定。

显然，Chandler 还没完成，甚至离完成还早。不过，既然人们已经用它进行日程管理，它就已经前所未有地真实地活着了。开发者们到达了一处精神上的里程碑；他们至少可以把一大块“恼人焦虑感”抛在身后。

0101010101010101101011010110

软件以这种或那种方式完成后，就会拥有令人惊异的寿命。尽管做个好程序可能需要无穷多时间，但好程序有时也会存在同样长时间；这是软件时间的另一维度。用风靡于 20 世纪 60 年代的主流商业编程语言 Cobol 编写的程序，直至 20 世纪 90 年代后期仍在广泛使用。当时为了节省紧缺的内存空间，许多 Cobol 程序只用两位数字保存年份；当 2000 年——“Y2K”——日益逼近，就得加以修正或干脆废弃。

更近一些的软件历史上有许多这样的例子，某个程序在历尽艰难后又风行起来，或者濒临湮没时又咸鱼翻生。米奇·卡普尔继莲花公司后创建的第二间公司 On Technology 只制造出卡普尔期望软件的一小部分，但其 MeetingMaker 小组进度管理程序却在之后二十年里得到广泛采用。史蒂夫·乔布斯于 1985 年被驱逐后开办的 NeXT 公司满怀大志，虽然没能改变世界，甚至没能在商业或消费者市场赢得显著占有率，但 NeXT 系统却是今天麦金塔 OS X 操作系统的基础。

还记得我在 Salon 的同事于 2000 年搭建的小内容管理系统吗？——

它灾难般的部署过程让我备受折磨，也推动我着手做这本书的调研工作。这套系统后来变成了名为 Bricolage 的开源项目，为世界卫生组织（World Health Organization）、兰德公司（RAND Corporation）[①]和霍华德·迪恩（Howard Dean）[②]总统竞选班子所采用。

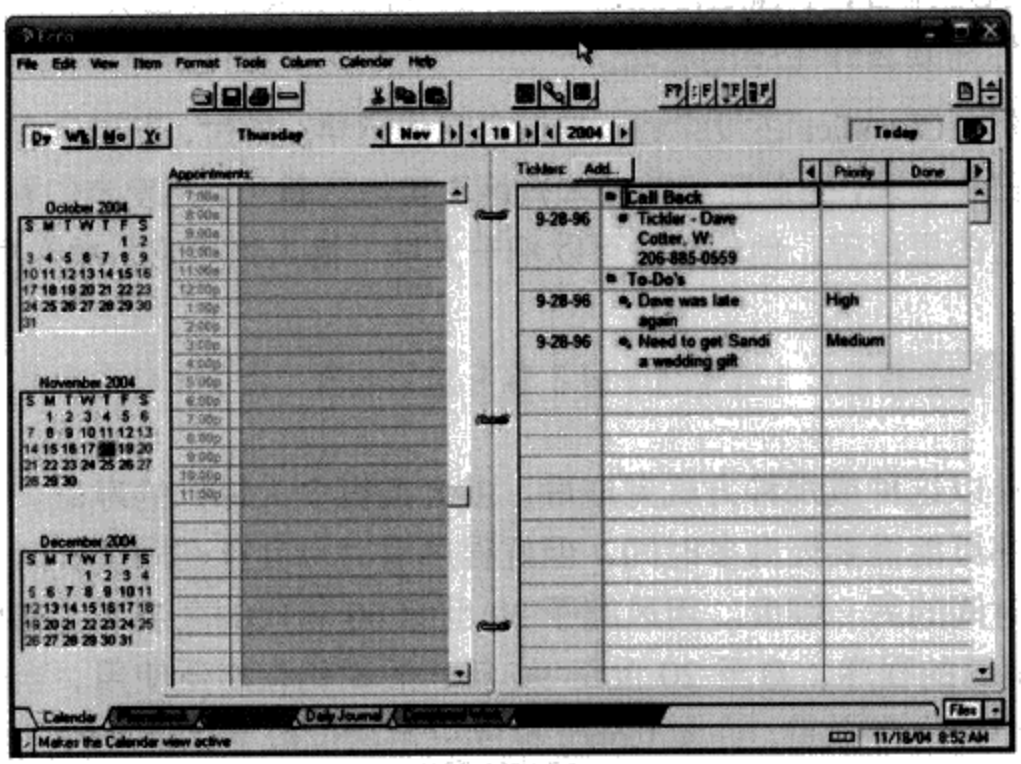


图 11-3 被开发者放弃的 Ecco Pro 软件

你手中的这本书，其全部研究工作的组织、汇整、纲目、交叉引用以及无穷无尽的调整也是用一套于 1993 初次发布的 Windows 程序 Ecco Pro 来做的。1997 年，Ecco 成了“孤儿”，因为其所有者认为他们无法跟 Microsoft Outlook 竞争（尽管它让 Outlook 那死板的设计看起来像是——喔，不提也罢！），而 Ecco 的代码在之后将近十年中再也无人问津。它在我的 Windows 计算机上运行良好，而且从来不会丢失我这些年托付给它的数据。它不能跨平台，不开源，也不基于 Web。但它在概括和灵活组织（Chandler 承诺但还没交付的功能）方面要比我找到的其他程序

[①]美国非盈利性的研究和咨询服务机构。总部设在加利福尼亚州的圣莫尼卡。兰德为 Research and Development（研究和发展）的缩写 RAND 的音译。前身是 1946 年接受为美国空军进行研究任务的“兰德计划”。1948 年 11 月正式成立并以兰德公司命名。以研究军事尖端科学技术和重大军事战略而著称。

[②]2004 年总统大选候选人，采用了新传媒战略，后落败。

干得好。我很高兴地发现，在 OSAF 认识的一些程序员中，颇有几位也是 Ecco 的拥趸。

老程序很少拥有新潮的图形界面或风行一时的特性。但它有一种不可低估的优势——以工作为取向。适当使用的程序就像是精心打理的旧花园，或者轻柔弹奏的老吉他：其粗糙边缘已然锉去，缺陷已被发现和修正，人皆知其表现物有所值。



我写到 2006 年中的时候，Chandler 的未来——数百万用户？沦为谈资？——尚无法预测。我们只能打赌它的故事估计将继续遇到不可预知的转折。种瓜得豆，世间之常情也。软件项目乃意外结果律的绝佳演绎，其创新和突破也常是计划之外的副产品。

例如：沃德·坎宁汉的 wiki 之所以能问世，不是因为程序员本来就要做一套谁都能添加的网页程序，他只是想让协同者们能够自行修改错误罢了。那些在 1999 年开办了小公司 Pyra 的开发者们致力于生产协同项目管理工具；他们也打造了一个写网页日志、与用户沟通的小程序。那个小小的子项目变成了 Blogger——世上最大的 blog 托管服务，拥有几百万用户（Google 于 2003 年收购了它）。软件公司 Ludicorp 设计了一套“大规模多用户在线角色扮演游戏”的原型系统；这个叫做 Game Neverending 的项目很快就寿终正寝，但其创建者们用它的一些部分修修改改，搭建了 Flickr 相片共享服务，在 2004 年成为大热网站（雅虎公司于 2005 年收购了 Flickr）。

就此而论，谁又能料到，某个欧洲量子物理实验室中一位程序员为了方便共享研究成果，在 1991 年做的小小尝试，会演化成今天的万维网？

有些观察者已将一项重要的意外成果归功于 Chandler。他们认为，Chandler 是否能做成一套有用的程序、或者达成任何最初目标都不重要；重要的是在搭建自己的开源组织时，卡普尔认识了 Mozilla 的米歇尔·贝克，而且，当 AOL/Netscape 决意不再支持这个开源浏览器项目时，他适逢其会地解救了它，为 Mozilla 基金会接生，将 Firefox 放置到成功的轨道上。行文至此时，Firefox 已平地赢得大约 10%的浏览器市场（本书付印时[①]超过 1 亿 7000 万下载量），其竞争迫使微软转头料理久已闲置的 Internet Explorer 产品，并重启浏览器开发业务，从而让散布各处的网页用户受益。

[①] 本书英文版于 2007 年 1 月出版。

Chandler 启程之时，卡普尔或任何其他人都不可能预见此事。以后肯定还会有更多类似的事情发生。

01010101010101011010110110

到 Chandler 开发者们准备于 2005 年 11 月发布 Chandler 0.6 版、满心虔诚地“吃狗食”的时候，他们已经庆祝了 OSAF 成员的四次生日和三次婚礼。他们迎来了几十位新同事，也送走了几乎同样数量的老伙伴。他们编写了数以万计的代码行。他们记录了几千个缺陷，修正了其中相当部分。他们也面对了自项目初次构思以来快速变化的软件市场。



图 11-4 庆祝 Chandler 0.6 版推出

微软最新的 Windows 版本在历尽延误的磨难后，将发布日期推迟到 2007 年初，此正当其成形之时。它已摘去 Longhorn 的代号，取名为 Vista——具有讽刺意味的是，这正是安迪·赫兹菲尔德给 Chandler 早期原型取的名字。Windows Vista 也承诺过类似 Chandler 那种对计算机用户与他们的个人信息之间关系的修补；现在，虽然微软花去 5 年时间，它看起来却更像是个升级版。在微软一再延误、Chandler 拼命找北的时候，基于 Web 的软件生态系统业已爆发：2005 年，好几十个小规模的新公司开始向互联网用户提供互相竞争的个人信息管理服务——包括 6 个基于 Web 的日历服务在内。

Chandler 0.6 版发布前，我和卡普尔交谈过，想知道他是否会觉得这样的情形令他沮丧。Chandler 真能与众不同吗？“绝对如此，”他答道，“0.6 版已经有了多个潜在的‘杀手级特性’。其中最重要的就是‘读/写共享’——让人们能轻易地发布日历，好让别人可以**读取和编辑**。Chandler 对数据的处理给共享增添了一些迷人的特性：比如，一对繁忙的夫妻互相共享日历，像卡普尔和他妻子那样，从其中一个日历取出条目、添加到配偶的日历，再修改自己日历上的条目，修改就会传递过去，在两个日历间同步。“这是我们独有的功能！”卡普尔高呼。

他说对了：如他所言，这“真酷”。不过离 Chandler 早期立下的改变世界的宏愿还有漫漫长路要走，而且卡普尔也承认自己感到“有些受挫”。

我问他，假使能搭乘时间机器回到项目开始时，会给当时的自己什么建议。“这是个 20 人的项目，”他答道。“不是 1 个人、2 个人或 3 个人，也不是 300 人。那宏愿需要中等规模的团队来完成，而当拥有中等规模的团队时，各种影响也随之而来。要进行大量协调。我们还在学习，而且要继续学下去。如何执行？如何设定目标？我从没做过这类涵盖这样的范围、20 人团队的实践项目，所以好多事都让我惊奇，而且会继续

吃惊下去。"

"我们一直过度投入在架构和设计上，其成果在下一个、甚至两个开发周期中都体现不出来——也就是说，后 6 个月或 12 个月都没戏。你要付不复敏捷的代价。我建议你做更多我们在过去几年来一直在做的事，即排列创新顺序、给工作分级、不要太有野心——除非这样做才能符合明年目标的要求，否则就别做 CPIA 那样的架构。我越来越觉得，此处之道在于做敏捷开发而又不放弃愿景——而且，坦率地说，我甚至没那样定义要着手解决的问题。"

有了这么多次令人气馁的延误，他有没有想过干脆叫停 Chandler 呢？"没有。我曾经非常沮丧过。但我从自己的经历中学到，如果感到自己太过绝望，应该置之不理，避免采取行动——因为这种感觉往往是暂时的，多数只是见景生情。我反而会出去一段时间，一个小时、几个钟头、一天或两天。取决于何时能够重新振作精神。每次我回来时，都对大家说，我相信咱们能找到法子修正路线、实现愿景。"

我说，看来 OSAF 最终还是接受了 6 个月的发布周期；在项目早期，他们总想设定比这短得多的期限。

"那只是一厢情愿——太过一厢情愿，"他若有所思地说，"我想，不切实际之举毫无助益。如果你能找到一种应付真实境况而又不致受挫或阻滞的有效方法，不妨直面现实之痛。我们还在继续这么做，而且不会就此罢手。开发软件仍属难事。自我起步于今已有 20 年，它可丝毫没变容易——你可以多下力气，但本质上都会遇到同样的协调问题。那就是我们存身的宇宙，所以得进行适当的调整——把食物切成小块，细嚼慢咽。消化掉。然后再吃下一口。"

"我真的觉得，"他补充说，"我们学到了大量关于如何开发软件的组织和个人经验。我们像是在重新发明轮子——这些都是别人已然知晓的

事，所以我们花了稍长的时间来学习。不过，学到手之后，它就是一种知识资本，而我也打算坚决实行之，而且要坚持到底。”



Chandler 开始于一个愿景——让 Agenda 之魂在当代程序中重生，消除地窖、让用户能随己所愿任意切合数据。这一愿景的初衷简单而优美。但软件开发总是把简单优美的愿景细细分解，分成数以百万计的实现细节、接口选择和妥协折中。在某种意义上，做好软件的艺术就在于以一种能复原的方式细分愿景——就像那些在互联网上传递、又在你计算机上重组为连贯信息的数据包一样，或者就像是企业号飞船（Starship Enterprise）[①]的乘员，在传送室被分解为粒子，又传递到某个星球表面、重塑人形。

2005 年底，Chandler 也穿过了那道光柱，开始逐渐闪现。三年前卡普尔初次对外宣布 OSAF 时聚集在 Chandler 邮件列表上的软件狂徒和开源拥趸们已命驾他方，其中许多人还断言 Chandler 走入死胡同或成为后车之鉴。Chandler 会倾覆在路旁，任呼啸而过的 AJAX 大军践踏吗？还是它会像 Mozilla 一样，数年在野之后，重新赢得大众的心并占据桌面？

对于 Chandler 的成功我抱有私心。在经历了数次操作系统升级或计算环境的偶然改变后，我的 Ecco 程序在某天终于不能再运行。我刚开始尝试用 Chandler 做日历管理；在准备好写下一本书时，要能用 Chandler 来组织我的调研成果就好了。在我的 Chandler 日历上，这个日子赫然在望。

是挥别 OSAF 和 Chandler 的时候了，不仅是因为我有稿子要写。关于为什么做软件如此之难，还有怎样让它变得稍微容易些，卡普尔及其团队学到了很多；所以，我希望我们也能有所收获。但 Chandler 自身那

[①]美国电视系列剧《星舰迷航》中的宇宙飞船。

尚未结束的故事，我却要弃之不顾了——因为，无论是好是坏，它都没有终点。

到本书付印之时，OSAF 可能又已发布了两个版本。你可以访问其网站，如果你是程序员的话，还可以下载源代码。也许你只想运行“最终用户安装”版本，看看项目进行到何种程度。它是陷入软件时间的另一个循环了呢，还是已经实现做点什么大事的“模糊前景”、鼓舞社区开发者加入其中并使其日益壮大？

兹录一事：在我写作本章时，有个图标闪动起来、提醒我有新邮件。我点开邮件程序，原来是发到 Chandler-dev 邮件列表的一封邮件，作者是戴夫·卡布兰尼克（Davor Cubrannic），英属哥伦比亚维多利亚大学（University of Victoria）的程序员，长期挂在 Chandler 邮件列表上。他建议：“我是 Ecco PIM 迷，它有一个很有用的 UI 特性，用户可以点击模拟时钟下拉框[像钟面的菜单]选择时间……所以在 Chandler 里通过文本控件输入时间感觉像是退步了，我用 wxPython 实现了 Ecco 的时钟下拉框。”

卡布兰尼克的邮件不只是提个想法；还附上了 180 行 Python 实现代码。“代码在 Windows 和 wxPython 2.6.1.0 上测试通过，”他写道。

泰德·梁回复说：“谢谢提交代码！我在 Mac 和 Linux 上试运行了。在 Linux 上工作正常。”但在 Mac 上会导致错误。

阿列克·福莱特也做了热情回复，说 0.6 版来不及加上这个新特性了。

赫基·托伊沃伦（Heikki Toivonen）建议卡布兰尼克把它登记到 Bugzilla，其中有一个特别的分类，专用于填写更像是“增强”而不是缺陷的特性。

卡布兰尼克将其标记为第 4520 号缺陷：“提供在事件细节视图中输入时间的替代方式。”

尾　声　长赌

[2005 年～2029 年及以后]

就在某处，某人的拳头又在重击桌面了。我们怎么就是不能像造桥那样做软件呢？

嗯，也许我们已经这样做了。

当 OSAF 的程序员们奋力搭建他们的代码塔，往数据资料库上堆 Python 代码、再往上堆缺陷修正代码时，我也关注着海湾大桥新东侧部分的进展。该项目始于 20 世纪 90 年代，最初投资十亿美元多一点，计划替换那道长 4.5 英里、每天有数十万辆车通过的双层桥的东半侧。原本设计为一条低矮朴实的堤道，但出于政治竞争和地方荣誉感的需要，最后采纳了更为宏伟和独特的设计方案。新桥，一座“自锚吊桥”，将由单根桥柱牵拉。网一般的钢缆从那独柱顶端延伸下来，穿到路面以下、再折头挂到柱顶，往复交错、蔚为大观。它将不只是一座可以远观的漂亮大桥；它还是一座拥有大胆创新概念的大桥，一座自提靴带的桥——一座与侯世达心心相印的自引用大桥。

只有一个问题：以前从没造过类似的建筑。谁也不想承接工程。加州做招标时，唯一的竞标承包商开出了远远超出预计的价钱。

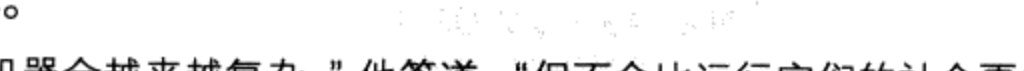
2004 年 12 月，加州州长阿诺德·施瓦辛格（Arnold Schwarzenegger）插手叫停项目，宣布湾区应承担更多项目膨胀的费用，并要求重审大桥的设计方案。别再管在已建成的半座桥上花的功夫，那海水环抱的高架桥体超过一英里长，汽车将由此慢慢爬上未来的主桥；每天早晨，你都能看到汽车爬上一条临时匝道、开到新路基上。施瓦辛格想要让项目方案回归实用和低成本。州长、州立法机关、州交通厅和当地政府花了好几个月时间讨价还价。交通厅称每延误一天就要多损失 400,000 美元。最后，2005 年 6 月，达成了一项折中方案，沿用较新奇的单塔式设计，藉由收取过桥费和其他方式来支付建筑费用，新设定的建桥工作完成时间：2012 年——自 Loma Prieta 地震撼松旧桥之后几乎四分之一个世纪。



图 1　建设中的海湾大桥

看着关于这场论战的材料，我不由得想到那些借严格流程和土木工程中屡试不爽的标准对编程梦想家们加以当头棒喝的软件管理手册。“软件开发需要更多规程，”他们会说，“桥造好一半后没人会去改动设计方案！”

加利福尼亚州成功地反驳了这种观点。



我们为什么不能像造桥那样做软件？当然不是说加州那种造桥法；发问者总在梦想一种基于可信公式的严格规程。除非我们能设计出一种与现实世界中支配工程学的质量、运动、能量计算范式相匹配的“软件物理学”，否则大概还是做不到。话虽如此，软件开发的种种技术困难还是在持续的改进面前步步败退。如同在造桥世界中发生的那些事一样。今天，我们把在这个领域的能力视为当然之事；全然忘记了（或者避而不谈）从 19 世纪到 20 世纪间造桥失败和桥梁坍塌的漫长历史，那时，新技术和尚未成熟的施工方法造成的灾难与损失，足以让我们领域中的软件灾难相形见绌。

随着技术流程的改进，我们错以为问题已经解决。但围绕技术流程的人类流程却仍旧十分棘手。设若真如弗里德里克·布鲁克斯所言，“做软件的难处在于决定说什么，而不是说出来，”则编程模型与造桥有多接近已无关紧要。造桥者在“决定说什么”方面稍强于做软件者。可他们也是人。

在 2004 年的一次开源大会上，科学家弗里曼·戴森（Freeman Dyson）被问及人类创造的技术系统会不会变得越来越复杂，复杂到连人类都无法理解。

“机器会越来越复杂，”他答道，“但不会比运行它们的社会更复杂。”

当然，编程领域能够、也将继续改进其技术能力和工具。程序员向来乐于自扫门前雪；他们不太可能会停止打磨算法、改进后台代码、调优开发平台。

然而，这就是在磨斧头——或者给牦牛剃毛。他们没触及问题的根本。实现人类世界中某个功能的任何机制迟早都要脱壳而出、与人交流。“究其根本，信息系统仅在与人接触时才有所价值，”杰隆·拉尼尔如是说。而一旦与人接触，对完美的追求就消散于无形了。

软件之本质困难，乃是强加于技术进步的人类自由意志和不确定性的通行费。尽管计算学科的兴起让许多科学家借其透镜审视人类机体——例如，假想人脑的功能映射到计算机结构上——而个体的人却仍旧不可被编程。人们之所为，就连最具想象力的程序员也无法逆料。人们之所愿，程序员们更不可预期。

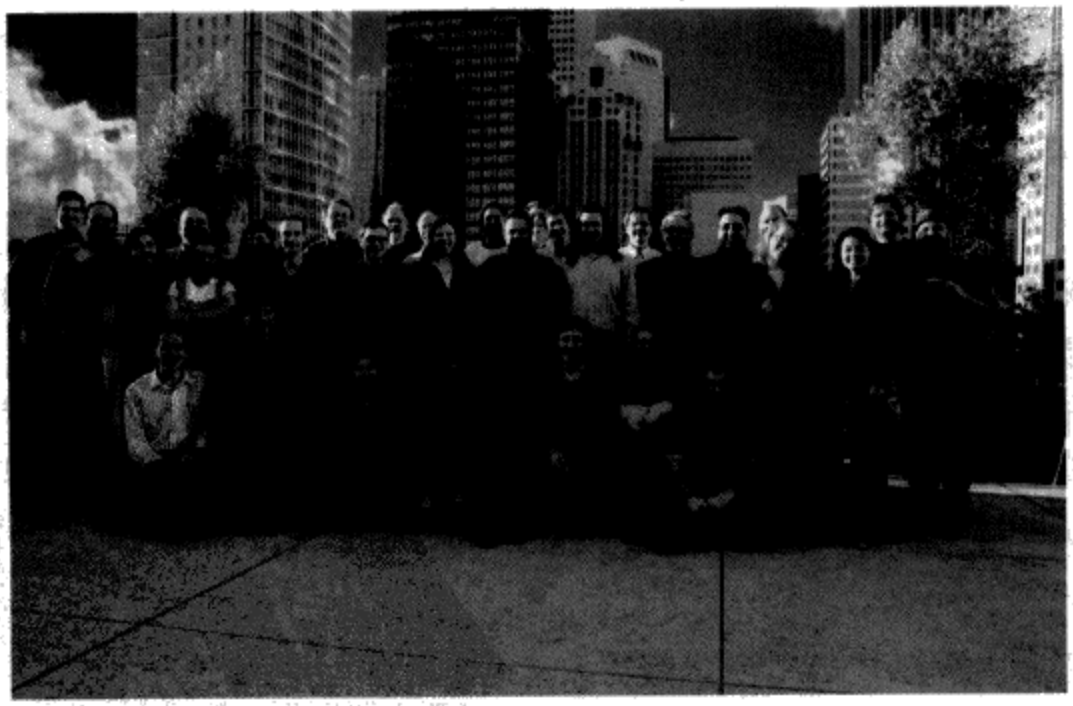


图 2　2006 年的 OSAF 全家福

2005 年夏天，比尔·盖茨在新加坡的微软研讨会上阐述了该公司实验室“计算机植入”项目的研究成果。这类电子假体首先将用于生产帮助盲人和聋人的产品。

此类研究无疑能结善果。为医疗应用制造可靠的数码设备当然是我们力所能及之事。但试图在人体中植入通用的计算机就会立刻招致质问和愤怒。我们该把编写软件的工作托付给谁？谁可受到信任修改软件？我们会愿意终此一生持续升级和修正缺陷吗？

盖茨，这位软件征服现代生活的代言者，看来也有这样的疑虑。“有个微软员工……总对我说‘我准备好了，接入电脑吧。’”他在新加坡的研讨会上说，“我却不敢苟同。我情愿与计算机两不相干。”

1010101010101010101101011010110

以我三年的软件时间经验，尽管也有令人激动的新方法论、清规戒律以及突破性的想法，做软件仍然真的很难。然而，我也能听到异议说，困难有不同类别。养孩子难。葬先亲难。出生和成长；与人同住或无人相伴；尝试去爱；无法去爱；接受他人之死；接受自己之死——都很难。软件？那是另一个困难世界，不太难的那种。

对此只能作答，对。实际上，对于那些乐意修补高耸的代码迷塔的人来说，与奇客们所谓“湿件[①]”生命的那种不可捉摸又冥顽不化相比，软件所呈现出的困难反而是某种欢迎。它自足。它理性。它不会心存怨恨。在这种意义上，软件一点都不难。它是一阵微风，用艾伦·乌曼在以编程为主题的小说《缺陷（The Bug）》中的话来说，它能使人“免于

[①]指人脑或人类思维。

世间的混乱”。

由于其重复的周期和无限期的延误，编程工作总让人想到薛西弗斯的劳役，没完没了地推巨石上山。典型的做无用功。薛西弗斯神话最知名的现代解释，阿尔伯特·加缪（Albert Camus）[1]的存在主义文本，告诉我们那受诅咒的希腊人在劳役中体验了一种奇异的愉悦。“你该想象得到薛西弗斯的快乐，”加缪说。

[1]法国存在主义文学家，1957年诺贝尔文学奖得主。

我在研究过程中访问和认识的多数程序员始终如一地、而且有时毫无来由地对工作持有乐观态度。如果他们是薛西弗斯的话，也会是快乐的薛西弗斯。

10101010101010101101011010110

一切倒塌又得以重建，
再造它们的人满心欢喜。

——威廉·巴特勒·叶芝（William Butler Yeats）[2]，
《天青石雕（Lapis Lazuli）》

[2]爱尔兰大诗人，1923年获诺贝尔文学奖。

计算机于20世纪50年代和60年代，也就是计算时代开始那些年，以奇迹般绝无错误的形象进入大众心中。错误属于人类；计算机不会犯错。只有那些与计算机近距离工作的人才明白，真实情况远比这复杂得多——在任何一张地图上，人类的错误与计算机的精确之间都没有可以辨别的边界；两者互有交错；尽管计算机在如此之多的方面卓立不凡，但还是没有也不能自行超越其创造者的能力与局限性。

最近几十年来，我们都得开始与计算机近距离工作，计算机从不犯错的神话就此失色。今天，我们开始了解到，世界赖以运行的软件并不比搭建和编写它们的人的双手和头脑来得完美。关于如何把软件做得更

好，那些人还有很多东西要学。但做得特别好、彻底地好——好到我们甚至不再意识到其存在；好到我们终于可以说，对于某种具体工作或活动，软件“做到头了”——这样的目标看来很难做到。

这可能会打击到程序员们，让他们感到无望。其实不然。为做软件付出的每分努力最终都注定是不够的，这种思路中有着某种自由度。它意味着总有进一步努力的空间，新一代总有机会来啃咬难题，某地总有某人欢迎“另一个”程序。



2002 年春天，正值米奇·卡普尔和 Chandler 团队的早期成员们着手搭建新软件的架构之时，卡普尔因为另一件事上了科技新闻头条：他加入了一场关于人工智能进展的“长赌（Long Bet）”。Long Bets（长赌）[①]是由非赢利组织永今基金会（Long Now Foundation）主办的项目，该基金会的创始人是《全球目录（Whole Earth Catalog）》杂志创办者史都沃·布兰德（Stewart Brand）和一群数字时代名人，旨在刺激有关长期议题和难题的讨论和创造性观念。作为该项目闪亮开幕的一个长赌，卡普尔下注 20,000 美元（赌赢所得尽归非盈利机构），打赌[②]到 2029 年不会有计算机或“机器智能”能通过图灵测试。（所谓通过图灵测试，一般是指计算机程序经由即时通讯工具骗过人类、使之相信正与人对话。）

与卡普尔对赌的是雷·卡兹维尔（Ray Kurzweil），一位在电子乐器和语音识别方面卓有成就的多产发明家，后来成为未来主义积极派的促进者。坚信机器能通过图灵测试，不过是卡兹维尔大信仰中的一部分——他认为人类历史将会被以指数级加速的摩尔定律及其他类似的直攀曲线驱赶到运转过载的程度。卡兹维尔认为，随着计算能力、存储能力和网络速度大变魔术，价格持续下降，直到可在技术上仿效人脑、用计算机

[①]该项目网址为：http://www.longbets.org/。

[②]卡普尔与卡兹维尔的对赌参见 http://www.longbets.org/1。

软硬件对人体自身的处理器进行反向工程时，我们就会遭遇危急时刻。同时，生物科技及其侍女——纳米科技——将以同等的爆炸性速率迅速进步。

当这些改变发生之后，我们就会到达卡兹维尔——及与他有着同样看法的人——所说的“奇点”时刻。这个词借自物理学，最初是由科学家和科幻小说作者维纳·温奇（Vernor Vinge）传播给大众；如同黑洞和从时空概念中借用的其他术语，奇点是对连贯性的破坏，是一个过去时间的终结点。在这个点上，一切都改变了，之后的情形不得而知。

卡兹维尔预测说，人工智能将带来人类历史上的奇点。到了21世纪20年代的某个时候，当其发生之时，人工智能通过图灵测试将见证奇点来袭——他说，那将会在21世纪40年代造成“人类体验的根本改变”。

乌托邦？不尽然。卡兹维尔小心地列出了这前景黯淡的一面。浩劫？谁知道呢——根据定义，奇点的结果不能为我们这些前奇点人所想象。大事？那是当然。

很容易拿卡兹维尔的数字末世论的怪异一面来开玩笑。他每天吃200片药的生命延长计划——维持他那50来岁的湿件，直至奇点之后的医疗进展开启不死之门——听来更像是某种深夜电视广告，而非严肃的科学家所为。但卡兹维尔在科技未来观察上迄今为止都足可信赖；他出言端肃。当他说“短期内我们总低估其难度，长期而言我们低估了其变化之大”时，历史站在了他那一边。

不过卡普尔认为他大错特错——整个“强大人工智能”项目，自20世纪50年代至今试图用硅晶和代码复制人类智慧的努力纯属闹剧一场。在一篇关于长赌下注的文章中，卡普尔写道，整个人工智能领域都误解了——和低估了——人类的智力。

作为人：

我们是具体化的生物；肉体性是我们的基础，肉体性也以无数种不同方式定义了我们的存在。

我们与周边环境紧密相连；在经验形成的过程中，感知环境和与环境互动具有等同的认知作用。

情感和认知同等基础，或更加基础；强烈和细微的感觉塑造了思维的封套。

我们是有意识的存在，既能外观亦能自省；精神人格和超越自我（用不太沉重的词来说）是我们能预与其中、也是我们所具有的境界。

卡普尔写道，计算机如何才能模拟在如此多维度上延展其存在的人类——或愚弄一个能够“探查其与典型人类沟通的能力”的人呢？

“最终，”他宣称，“我认为雷要比任何机器更聪明、也更有本事。”



卡普尔的观点与普遍看法非常一致（这也极大地佐证了他的观点）。还要过四分之一世纪我们才知道谁对谁错。

到那时，我该已经可以在新湾桥上行驶了。Chandler 或许成了繁荣的开源生态系统，也可能被埋葬在死代码的墓地中。而且或许，只是或许，世上程序员的大脑、才干和创造性将能找到一种摆脱软件时间流沙的方法。

不过我没下赌注。

2005 年，在和卡兹维尔打下长赌三年之后，也是在攀爬真实世界软

件开发壕沟多年之后，米奇·卡普尔加大了赌注。

“我要马上加一倍赌注，”他说，“我还没遇到支持他的人。现在，我不跟奇点泡泡里的人谈话。不过那些软件实践者，不管是 25 岁、45 岁还是 65 岁——都没压注在他那边。”

那似笑非笑的神情——也许纯属玩笑，或者蕴含了辛苦得来的经验——在他的眼中洋溢。“你越懂软件，就越不会去做软件。”

译后记

2007 年初，博文视点的郭立总经理打电话给我，说有本书想请我评估。拿到 Dreaming in Code 英文原版，读完第 0 章，我知道，这书无论如何是舍不得给别人做了。

软件乃是人类自以为最有把握、实则最难掌控的技术。本书作者罗森伯格对 OSAF 主持的 Chandler 项目进行田野调查，跟踪经年，试图借由 Chandler 的开发过程揭示软件开发中的一些根本性大问题。说起罗森伯格，也算奇人一个。他创办的 Salon.com 网站是美国最著名的网上出版物之一。此人乃是一全才，编程写作两手都硬，最难得的是他在高屋建瓴般的大眼光之外，收放自如的好手段。本书是讲一事，也是讲百千事；是写一软件，也是写百千软件；是写一群人，也是写百千万人。我相信，任何一个在软件领域稍有经验的技术人员看完本书，必掩卷长叹：做软件难。

也因为罗森伯格文笔太好，翻译起来特别困难。在试译首章时，笔下还算流畅，颇以为可以很快译完全书。孰知再往后做，就迎头撞上拦路虎：古往今来的典故、软件领域的轶事、冷峻中饱含感情的笔调、有

如神来般的适时幽默、精当的遣词造句、尚来不及上词典的流行语……读书不妨囫囵咽，翻译怎敢随手抛？盖阅读之快感，快而后有感；翻译是为读者贡献读起来有快感的文字，读者要读得快（快速、快乐），译者必然是快不了的。在翻译过程中，责任编辑江立工作非常仔细，挑出了译文中许多不确之处，和她合作真的很愉快。

我一向认为，世间语言是相通的。计算机语言也是一种语言。做软件，是把人类用自然语言表达的需求，翻译为计算机能理解的语言。做软件难，做人类语言的翻译亦难，抑或更难，因为除开信息的表达，人类读者还会要求风格等几乎不可定量的元素。以信、达、雅三字为翻译效果的判断标准，实在有些模糊。我更倾向于“准确、完整、保持风格”的说法。“准确、完整”相对（只是相对）容易一些，而“保持风格”就难多了。自己写东西越好的人，越容易“看不见”原文风格。翻译诚然是一种创造，但弃原文风格于不顾、只管自己爽的翻译，只好称之为“译述”，自成一体，与原文未免有点风马牛不相及的意思。

所以，能翻译与自己平日写作风格相近的原文，是多么幸运的事！我翻过两本技术书，非逼着自己去适应原文风格不可，是做了对读者有意义而让译者感觉憋屈的事。这次翻译 Dreaming in Code，酣畅淋漓，仿佛酷暑天喝了杯冰冻酸梅汤的神气。键盘声响，是罗森伯格写了我想写的字，也是我说着罗森伯格要说的话。穷一年之功，翻译这本 300 来页的书，值了！

二次排版完成后，我也陷入了失语状态。跟着罗森伯格在软件丛林里转了一圈，回到现实中，原来米奇·卡普尔业已辞去 OSAF 主席一职，并且会在 2008 年内撤走投资。OSAF 全职人员也立即从 27 人减少至 10 人。凯蒂·帕兰特在 Chandler 项目官方博客上宣布了这条消息[①]，一时间外界议论纷纷。

没有人可以指责米奇·卡普尔自私，有篇评论[②]写得好：“他养活

① 消息参见 http://blog.chandlerproject.org/2008/01/08/osaf-transitions/

② 评论参见 http://blogs.cnet.com/8301-13505_1-9847739-16.html?part=rss&subj=news&tag=2547-1_3-0-20

Chandler 和 OSAF 长达六年之久，寄望于能做出令人激动的创新 PIM 套件……六年后，Chandler 仍无定形……（离开的）时候差不多了。”六年半时间，上百万美元，几十号顶尖高手，换来幻梦一场，此时最郁闷的当属米奇·卡普尔。在 OSAF/Chandler 行将就木的今天，本书正可做它的墓志铭。

“这里躺着一个野心勃勃的开源项目。它曾立志超越 Outlook，最后却无疾而终。许多程序员以心血养育它，惜乎全不见成效。它是温室中的花儿，有过绚烂的梦想，还未绽放即已枯萎。在那软件的花园中，还有多少会渐次凋零呢？”

韩 磊

2008 年 5 月于北京

附录 A 专有名词对译表

Scott Rosenberg　司各特·罗森伯格
A Brief History of Time　《时间简史》
A City Is Not a Tree　《城市不是树》
A Pattern Language　《模式语言》
Abby Mackness　艾比·麦克尼斯
Abraham Lincoln　亚伯拉罕·林肯
Accel Partners　Accel 合伙人公司
Advanced Automation System/AAS　先进自动系统
Aeron chairs　安乐按摩椅
Agile Manifesto　《敏捷宣言》
Agile Software Development　敏捷软件开发
Al Cho　艾尔·乔
Al Gore　阿尔·戈尔
Alan Cooper　艾伦·库珀
Alan Kay　艾伦·凯
Alan Turing　艾伦·图灵
Albert Camus　阿尔伯特·加缪
Alec Flett　阿列克·福莱特
Aleks Totic　阿历克斯·托蒂克
Alpo　爱宝牌
America Online　美国在线
Andi Vajda　安蒂·维达
Andy Hertzfeld　安迪·赫兹菲尔德
Anthony Baxter　安东尼·巴克斯特
Antioch　安提奥奇学院
Antonio Gramsci　安东尼奥·葛兰西
Aparna Kadakia　阿帕那·卡达基亚
Apocalypse Now　《现代启示录》
Apple　苹果公司
Arcadia　阿卡迪亚
Arnold Schwarzenegger　阿诺德·施瓦辛格
Asperger's Syndrome　艾斯伯格综合症/艾斯伯格症
Avalon　阿瓦隆
Baffin Island　巴芬岛
Bainbridge　贝恩布里杰
Barry Boehm　巴瑞·贝姆
Bay Area　湾区
Bay Bridge　海湾大桥

Belmont　贝尔蒙特
Bender　笨德
Benjamin Pierce　本杰明·皮尔斯
Bertrand Russell　伯特兰·罗素
Big-M methodologies　巨无霸方法论
Bill Atkinson　比尔·艾金森
Bill Joy　比尔·乔伊
Bjarne Stroustrup　比昂纳·斯卓思柯普
Board of Professional Engineers　专业工程师委员会
Bob Dylan　鲍勃·迪伦
Bob Marley　鲍勃·马雷
Bob Taylor　鲍勃·泰勒
Boehm's Law　贝姆定律
Booz Allen Hamilton　波茨·艾伦·汉密尔顿
Brad Cox　布拉德·考克斯
Brendan Eich　布仁丹·艾克
Brian Hayes　布莱恩·赫斯
Brian Kirsch　布莱恩·科斯齐
Brian Marick　布莱恩·马瑞克
Brian Moseley　布莱恩·莫斯利
Brian Randell　布莱恩·兰德尔
Brian Skinner　布莱恩·史金勒
Bryan Stearns　布莱恩·斯特恩斯
Business Week　《商业周刊》
C3PO　深思皮奥
Calendar Swamp　日历沼泽
Capability Maturity Model, CMM　软件成熟度模型
Carnegie-Mellon　卡耐基-梅隆
Chad Dickerson　查德·迪克森
Chandler　钱德勒
Chandler Presentation and Chandler Interaction Architecture　外观及互动架构
Charles Simonyi　查尔斯·西蒙尼
Chartres Cathedral　查特尔大教堂
Chi-Chao Lam　林智超
Christie　克里斯蒂
Christopher Alexander　克里斯多夫·亚历山大
Chrysler　克莱斯勒公司
Claire Tristram　克莱尔·崔斯兰
Clay Shirky　克莱·舍基
Communications of the ACM《美国计算机学会通讯》
Computers Ltd.　《计算机公司》
Cymbeline　《辛白林》
Dan Bricklin　丹·布瑞克林
Dan Gillmor　丹·基尔默
Dan Shafer　丹·沙弗尔
Dancing Rabbit Ecovillage　舞兔生态村
Daniel Dennett　丹尼尔·德奈特
David Allen　戴维·艾伦
David Cook　戴维·库克
David Gans　戴维·甘斯
David Gelernter　戴维·格林特
David Harel　戴维·哈瑞尔
David Heinemeyer Hansson　戴维·海因梅尔·汉森
David Lorge Parnas　戴维·洛奇·帕纳斯
David McCusker　戴维·麦卡斯科
David Surovell　戴维·萨鲁维
Davor Cubrannic　戴夫·卡布兰尼克
Death March　《死亡之旅》
Defense Acquisition University　国防采办大学
Dilbert　呆伯特
Donald Knuth　高德纳

Donn Denman 唐·登曼
Douglas Engelbart 道格拉斯·恩格巴特
Douglas Hofstadter 侯世达
Dr. Seuss 苏斯博士
Eagels 老鹰乐队
edge case 边缘案例
Edsger Dijkstra 艾德加·狄杰斯特拉
Edward J. Belove 爱德华·J·毕洛夫
Electronic Frontier Foundation 电子先锋基金会
Ellen Spertus 爱伦·斯帕图斯
Ellen Ullman 艾伦·乌曼
Elvis Costello 埃尔维斯·科斯特洛
Eminem 阿姆
end-user application software 终端用户应用软件
Eric S. Raymond 艾瑞克·S·瑞蒙德
Eric Sink 艾瑞克·辛克
Esther Dyson 艾斯特·戴森
Esther Sun 艾斯特·孙
Extreme Programming 极限编程
Federal Aviation Administration/FAA 联邦航空管理局
focus 焦点
Francis Fukuyama 弗朗西斯·福山
Franklin Institute 富兰克林纪念馆
Freada Kapor Klein 弗丽达·卡普尔·克莱恩
Frederick Brooks 弗里德里克·布鲁克斯
Freeman Dyson 弗里曼·戴森
Friends 《老友记》
Futurama 《未来狂想曲》
Future Combat System 未来战役系统
Garmisch 嘉美善
geek 奇客
geek out 奇拗
generative programming 生成编程
George Santayana 乔治·桑塔亚那
Gerald Jay Sussman 杰拉德·杰·苏斯曼
Gerald Weinberg 杰拉德·温伯格
Getting Things Done 《搞掂》
Go To Statement Considered Harmful 《Go To 语句被认为有害》
Godel, Escher, Bach: An Eternal Golden Braid 《哥德尔、艾舍尔、巴赫——集异璧之大成》
Gordian software 戈耳迪软件
Gordon Moore 戈登·摩尔
Grace Hopper 格瑞丝·胡珀
Grant Baillie 格兰特·贝列
groupware 群件
Guardian 《卫报》
Guido van Rossum 圭多·范·罗萨姆
halting problem 停机问题
Hammett 哈迈特
Harvard Business Review 《哈佛商业评论》
Heikki Toivonen 赫基·托伊沃伦
Hemingway 海明威
Henry Ford 亨利·福特
Herman Melville 赫曼·梅尔维尔
Home Comforts 《让家更舒适》
Howard Dean 霍华德·迪恩
Howard Street 沃德街
In Search of Excellence 《追求卓越》
Intense Programming 激烈编程
intentional community 概念社区

intentional programming　意识编程
Intentional Software　意识软件公司
Internal Revenue Service　国内税务局
International Software Engineering Institute　国际软件工程研究所
Internet Engineering Task Force，IETF　互联网工程任务组
Internet Standards World　互联网标准世界
IT Doesn't Matter　《IT 无关紧要》
Ivan Sutherland　伊万·萨瑟兰
J. Edgar Hoover　J·埃德加·胡佛
J.Presper Eckert　J·普莱斯帕·埃克特
J.S.Bach　巴赫
James Bach　詹姆斯·巴赫
James Clerk Maxwell　詹姆斯·克勒克·麦克斯韦
James Fallows　詹姆斯·菲罗斯
James Gosling　詹姆斯·高斯林
James Noble　詹姆斯·诺博尔
Jargon File　《新黑客词典》
Jaron Lanier　杰隆·拉尼尔
Jason Fried　杰森·弗瑞德
Jed Burgess　杰德·伯吉斯
Jeffrey Harris　杰弗瑞·哈里斯
Jeremiah　耶利米
Jerry Kaplan　杰瑞·卡普兰
Jim Clark　吉姆·克拉克
Joel on Software　祖尔谈软件
Joel Spolsky　祖尔·索伯斯基
Joel Test　祖尔测试
John Anderson　约翰·安德森
John Backus　约翰·柏柯斯
John Brockman　约翰·布罗克曼
John McCarthy　约翰·麦卡锡
John Sculley　约翰·史考利
John Von Neumann　约翰·冯·诺依曼
Jon Ogg　扬·奥格
Jon Udell　扬·乌代尔
Jonathan Sachs　乔纳森·萨赫斯
Kaitlin "Ducky" Sherwood　“鸭子”凯特琳·舍伍德
Katie Parlante　凯蒂·帕兰特
Kent Beck　肯特·贝克
KURTZ　克兹
L. da Vinci/Leonardo da Vinci　莱昂纳多·达·芬奇
L. Peter Deutsch　L·彼得·德奇
Lamentations　耶利米哀歌
Lapis Lazuli　《天青石雕》
Larry Constantine　拉瑞·康斯坦丁
Larry Wall　拉瑞·沃
League for Programming Freedom　编程自由联盟
Learn This Month's Flavor of Programming Language in 17 Seconds　《17 秒学会本月流行编程语言》
Level Playing Field Institute　推平游戏场研究院
Linus Torvalds　李纳斯·托瓦茨
Linux kernel swear words　Linux 内核誓言
Linux Times　《Linux 时报》
Lisa Dusseault　丽萨·杜索特
literate programming　字面编程
Livermore　利福摩尔
Long Now Foundation　永今基金会
Lorne Greene　罗恩·格林
Lotus　莲花/莲花公司，作为产品名称时不翻译
Lou Montulli　罗·蒙特利

M.C.Escher 艾舍尔
Macintosh 麦金塔/麦金塔操作系统/麦金塔计算机
Macintosh 《麦金塔》杂志
Making Design Decisions: Some Principles 《设计决策：一些原则》
Mark Minasi 马克·米纳西
Market Street 市场大道
Marvin Minsky 马尔文·明斯基
Masterpiece Engineering 《杰作工程》
Matt Groening 马特·格洛宁
Maurice Wilkes 莫瑞斯·威尔克斯
McDonald's 麦当劳
MDA(Model Driven Architecture) 模型驱动架构
MDD(Model Driven Development) 模型驱动开发
meatspace 人肉空间
Melon Foundation 梅隆基金会
Melville 梅尔维尔
Menlo Park 门罗公园
Merlin Mann 默林·曼恩
Michael Toy 迈克尔·托伊
Microsoft 微软/微软公司，在产品中出现时不翻译
Mike "Codebear" Taylor “代码熊”麦克·泰勒
Mills College 米尔斯女子大学
Mimi Yin 尹咪咪
Mitch Kapor 米奇·卡普尔
Mitchell Baker 米歇尔·贝克
Mitchell Kapor 米歇尔·卡普尔
Moby-Dick 《大白鲸》
Mona Lisa 《蒙娜·丽莎》
Monty Python's Flying Circus 巨蟒的飞行马戏团
Morgen Sagen 摩根·萨奇
Neal Stephenson 尼尔·史蒂芬森
nerd 讷客
Netscape 网景/网景公司，作为产品名时不翻译
New York Times 《纽约时报》
Newton 牛顿
Nicholas Carr 尼古拉斯·凯尔
Nigel Tufnel 奈吉尔·塔夫内尔
No Silver Bullet 《没有银弹》
Norbert Wiener 诺波特·维纳
Oakland 奥克兰
Odin 欧丁
Oracle 甲骨文/甲骨文公司，作为产品名称时不翻译
Pac Bell Park 太平洋贝尔公园
Palm Beach 棕榈滩
Palo Alto 帕罗·阿尔托
Parthenon 帕台农神庙
Patrick Leahy 帕崔克·利亚
Patterns community 模式社区
Paul Graham 保罗·格雷汉姆
Personal Software Process，PSP 个体软件过程
Peter Drucker 彼得·德鲁克
Philippe Bossut 菲利普·博苏特
Phillip J.Eby 菲利普·J·艾比
phonotropic 表面互动
Piet Hein 皮耶特·海因
Pieter Hartsook 皮耶特·哈特索科
Portland Pattern Repository 波特兰模式资料库
Potemkin-village 波坦金村庄
Puget 普杰特海河
Python is Not Java 《Python 不是 Java》

queer　同性恋
R2D2　糊涂迪图
RAND Corporation　兰德公司
Raphael Finkel　拉斐尔·芬克尔
Rapid Application Development　快速应用开发
Ray Kurzweil　雷·卡兹维尔
Ray Ozzie　雷·奥兹
Raymond Chandler　瑞蒙德·钱德勒
Raytheon　雷神公司
Red Hat　红帽/红帽公司，作为产品名时不翻译
Revolution in the Vally　《硅谷革命》
Richard Gabriel　理查德·戈布瑞尔
Richard Stallman　理查德·斯托曼
Rob Reiner　罗德·瑞纳
Robert Biddle　罗伯特·毕多
Robert Britcher　罗伯特·布瑞切尔
Robert Glass　罗伯特·格拉斯
Ron Jeffries　罗恩·杰弗里斯
Rube Goldberg　鲁德·古德堡
San Jose Mercury News　《圣何塞水星报》
Scott Mace　司各特·麦斯
Scrum　争球式开发
Sears　希尔斯百货公司
Shakespeare　莎士比亚
Sheila Mooney　雪拉·莫内
Sisyphean　薛西弗斯的任务
Sisyphus　薛西弗斯
Software Aspects of Strategic Defense Systems　《战略防御系统的软件部分》
Software Design Manifesto　《软件设计宣言》
Software Engineering Institute　软件工程学院
Software Runaways　《软件开发的滑铁卢》
Space Invaders　《太空入侵者》
spiral model　螺旋模型
Standish Group　斯坦迪什集团
Stanford Research Institute　斯坦福研究所，缩写时不翻译
Star Wars　《星球大战》
Starship Enterprise　企业号飞船
Stephen Hawking　斯蒂芬·霍金
Stephen R Covey　史蒂芬·R·考维
Steve Jobs　史蒂夫·乔布斯
Steve Silberman　史蒂夫·希尔波曼
Stewart Brand　史都沃·布兰德
Strategic Defense Initiative，SDI　主动战略防御系统
Stuart Parmenter　史都沃·帕门特
Sun Microsystems　太阳微系统公司/Sun 公司/Sun
Team Software Process，TSP　小组软件过程
Ted Leonsis　泰德·莱昂西斯
Ted Leung　泰德·梁
Ted Nelson　泰德·奈尔森
The Art of Computer Programming《计算机编程艺术》
The Bug　《缺陷》
The End of History　历史终结
The Geek Syndrome　《奇客症候群》
The Human Dynamics of Information Technology Teams　信息技术团队中的人类变数
The Humble Programmer　《小程序员》
The Immaturity of CMM　《CMM 不成熟之处》
The Inmates Are Running the Asylum　《精神病患管医院》
The Law of Leaky Abstractions　《漏洞抽象法则》

The Limits of Software 《软件极限》
The Psychology of Computer Programming 《计算机编程心理学》
The Science of Computer Programming 《计算机编程科学》
The Seven Habits of Highly Effective People 《高效人士的7个习惯》
The Software Conspiracy: Why Software Companies Put Out Faulty Products, How They 《软件阴谋：为什么软件公司
Can Hurt You, And What You Can Do About It 《扔出错误重重的产品，它们如何能伤害你，而你又该怎么做》
The Sun Also Rises 《太阳照常升起》
The Wizard of Oz 《绿野仙踪》
Theory and Practice 《理论与实践》
Theseus 提修斯
Thirerry Bardini 希尔瑞·巴蒂尼
This is Spinal Tap 《摇滚万万岁》
Thor 托尔
Tim Berners-Lee 提姆·伯纳斯-李
Tim O'Reilly 蒂姆·奥莱理
Time 《时代》
Time Warner 时代华纳
Tom Peters 汤姆·彼得斯
Tom Simpson 汤姆·辛普森
Tower of Babel 巴别塔
Twelfth Night 《第十二夜》
Unified Modeling Language 统一建模语言
University of Victoria 维多利亚大学
USA Today 《今日美国》
Vannevar Bush 范纳尔·布什
Vernor Vinge 维纳·温奇
Vidar Holen 维达·霍伦
Vitruvius 维楚维斯
W. Edwards Deming W·艾德沃茨·戴明
Wall Street Journal 《华尔街日报》
Ward Cunningham 沃德·坎宁汉
water-cooler effect 冷水机效应
waterfall model 瀑布模型
Watts Humphrey 瓦茨·汉弗里
Werner Erhard 维纳·艾哈德
Whole Earth Catalog 《全球目录》杂志
Wikipedia 维基百科
WILLARD 威利亚德
William Butler Yeats 威廉·巴特勒·叶芝
William Wegman 威廉姆·维格曼
Wired 《连线》
World Health Organization 世界卫生组织
World Wide Web 万维网
Yahoo 雅虎公司
Yellow Springs 黄溪泉
Yertle the Turtle 《乌龟雅图》
Zeno's Paradoxes 芝诺悖论
Zeus 宙斯

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

“每个有志于开发畅销产品的程序员都值得耐心去品味这个故事。”

——CSDN总裁 蒋涛

“《梦断代码》是一本代码史。”

——DoNews制作人，千橡集团副总裁 刘韧

“《梦断代码》乃承Tracy Kidder《新机器灵魂（The Soul of a New Machine）》血脉之初见者，融技术眼界与叙事功夫于一炉，实多年未见之奇书。读毕，当可了解软件术士之所为。”

——詹姆斯·菲罗斯（James Fallows），《大西洋月刊（The Atlantic）》

“技术人员爱把复杂的问题形容为‘非凡’。司各特·罗森伯格选取了极度非凡的主题，并使之浅显易懂。他盛赞编写代码的人，但也坦承他们如常人般复杂而有缺陷。《梦断代码》实在是一流的报道和著作。”

——丹·基尔默（Dan Gillmor），公民媒体中心主席及《自媒体（We the Media）》作者

“《梦断代码》探索了编程如何鼓舞和破坏人类对新工具的创造，既令人迷醉，又使人冷静。本书文笔优美，专为对创造与革新之根源感兴趣者——无论是开发者还是其他人——所写。”

——史蒂芬·强森（Steven Johnson），
《开机（Everything Bad Is Good for You）》及《幽灵地图（The Ghost Map）》作者

“司各特·罗森伯格放胆踏入非程序员未敢涉足之地：那人类想象苦苦变身为代码的旋风中央。在《梦断代码》中，他华丽地将一家初创软件公司的故事与我们理顺编程过程的（无穷尽）努力结合起来讲述。”

——艾伦·乌曼（Ellen Ullman），
《缺陷（The Bug）》和《走近机器（Close to the Machine）》作者

更多详情请登录http://www.dreamingincode.cn/。

网上订购：www.dearbook.com.cn
第二书店·第一服务

责任编辑：江　立
责任美编：张子建　李　玲

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。

上架建议：软件开发>人文

ISBN 978-7-121-06679-5
9 787121 066795 >

定价：49.00元